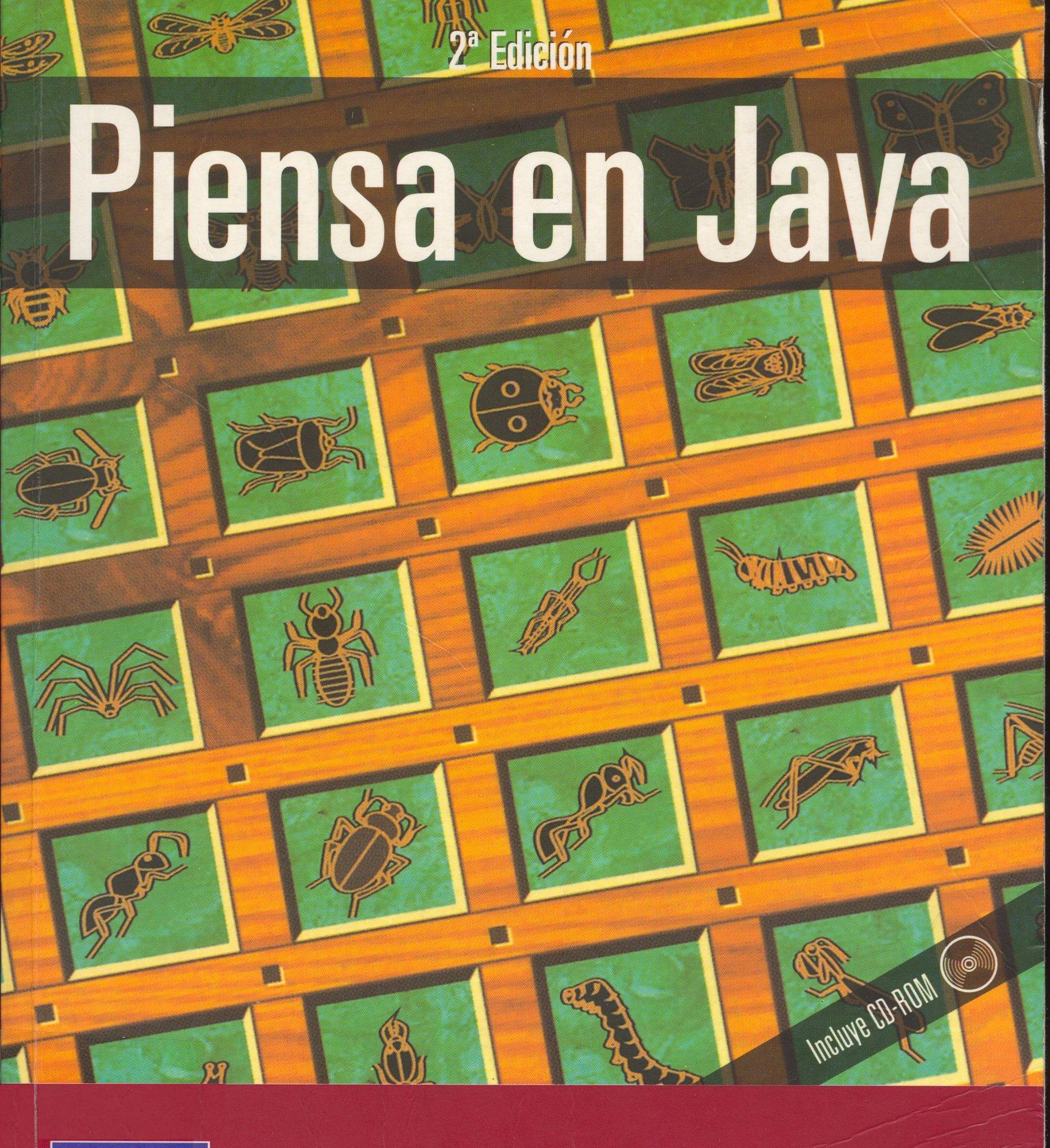


Prentice Hall

Bruce Eckel

Piensa

en

Java

Piensa en Java

SEGUNDA EDICIÓN

Bruce Eckel

Traducción:
Jorge González Barturen
Facultad de Ingeniería
Universidad de Deusto

Revisión técnica:
Javier Parra Fuente
Ricardo Lozano Quesada
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Universidad Pontificia de Salamanca en Madrid

Coordinación general y revisión técnica:
Luis Joyanes Aguilar
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Universidad Pontificia de Salamanca en Madrid



Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo
San Juan • San José • Santiago • Sao Paulo • White Plains

Datos de catalogación bibliográfica

Bruce Eckel
PIENSA EN JAVA
Segunda edición

PEARSON EDUCACIÓN, S.A. Madrid, 2002

ISBN: 84-205-3192-8
Materia: Informática 681.3

Formato 195 x 250 Páginas: 960

Bruce Eckel
PIENSA EN JAVA, segunda edición.

ISBN: 84-205-3192-8
Depósito Legal: M.4.162-2003
Última reimpresión, 2003

PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

Traducido de:
Thinking in JAVA, Second Edition by Bruce Eckel.

ISBN: 0-13-027363-5

Edición en español:
Equipo editorial:
Editor: Andrés Otero
Asistente editorial: Ana Isabel García
Equipo de producción:
Director: José A. Clares
Técnico: Diego Marín
Diseño de cubierta: Mario Guindel, Lía Sáenz y Begoña Pérez
Composición: COMPOMAR, S.L.
Impreso por: LAVEL, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos



A la persona que, incluso en este momento, está creando el próximo gran lenguaje de programación.

Índice de contenido

Prólogo

Sugerí a mi hermano Todd, que está dando el salto del hardware a la programación, que la siguiente gran revolución será en ingeniería genética.

Tendremos microbios diseñados para hacer comida, combustible y plástico; limpiarán la polución y en general, nos permitirán dominar la manipulación del mundo físico por una fracción de lo que cuesta ahora. De hecho yo afirmé que la revolución de los computadores parecería pequeña en comparación.

Después, me di cuenta de que estaba cometiendo un error frecuente en los escritores de ciencia ficción: perderme en la tecnología (lo que por supuesto es fácil de hacer en ciencia ficción). Un escritor experimentado sabe que la historia nunca tiene que ver con los elementos, sino con la gente. La genética tendrá un gran impacto en nuestras vidas, pero no estoy seguro de que haga sombra a la revolución de los computadores (que hace posible la revolución genética) —o al menos la revolución de la información. La información hace referencia a comunicarse con otros: sí, los coches, los zapatos y especialmente la terapia genética son importantes, pero al final, ésto no son más que adornos. Lo que verdaderamente importa es cómo nos relacionamos con el mundo. Y cuánto de eso es comunicación.

Este libro es un caso. La mayoría de colegas pensaban que estaba un poco loco al poner todo en la Web. "¿Por qué lo compraría alguien?", se preguntaban. Si hubiera sido de naturaleza más conservadora no lo habría hecho, pero lo que verdaderamente no quería era escribir más libros de computación al estilo tradicional. No sabía qué pasaría pero resultó que fue una de las cosas más inteligentes que he hecho con un libro.

Por algún motivo, la gente empezó a mandar correcciones. Éste ha sido un proceso divertido, porque todo el mundo ha recorrido el libro y ha detectado tanto los errores técnicos como los gramaticales, y he podido eliminar fallos de todos los tipos que de otra forma se habrían quedado ahí. La gente ha sido bastante amable con ésto, diciendo a menudo " yo no quiero decir esto por criticar...", y tras darme una colección de errores estoy seguro de que de otra forma nunca los hubiera encontrado. Siento que éste ha sido un tipo de grupo de procesos que ha convertido el libro en algo especial.

Pero cuando empecé a oír: "De acuerdo, bien, está bien que hayas puesto una versión electrónica, pero quiero una copia impresa proveniente de una auténtica editorial", puse mi mayor empeño en facilitar que todo se imprimiera con formato adecuado, pero eso no frenó la demanda de una versión publicada. La mayoría de la gente no quiere leer todo el libro en pantalla, y merodear por un conjunto de papeles, sin que importe cuán bien impresos estén, simplemente no era suficiente. (Además, tampoco creo que resulte tan barato en términos de tóner para impresora láser.) Parece que a fin de cuentas, la revolución de los computadores no conseguirá dejar sin trabajo a las editoriales. Sin embargo, un alumno me sugirió que éste podría ser un modelo para publicaciones finales: los libros se publicarán primero en la Web, y sólo si hay el suficiente interés, merecerá la pena pasar el libro a papel. Actualmente, la gran mayoría de libros conllevan problemas financieros, y quizás este nuevo enfoque pueda hacer que el negocio de la publicación sea más beneficioso. Este li-

bro se convirtió en una experiencia reveladora para mí de otra forma. Originalmente me acerqué a Java como "simplemente a otro lenguaje de programación", lo que en cierto sentido es verdad. Pero a medida que pasaba el tiempo y lo estudiaba más en profundidad, empecé a ver que la intención fundamental de este lenguaje es distinta de la de otros lenguajes que he visto.

La programación está relacionada con gestionar la complejidad: la complejidad del problema que se quiere solucionar, que yace sobre la complejidad de la máquina en que se soluciona. Debido a esta complejidad, la mayoría de nuestros proyectos fallan. Y lo que es más, de todos los lenguajes de programación de los que soy consciente, ninguno se ha lanzado completamente decidiendo que la meta de diseño principal fuera conquistar la complejidad del desarrollo y mantenimiento de programas[1]. Por supuesto, muchas decisiones de diseño de lenguajes se hicieron sin tener en mente la complejidad, pero en algún punto había siempre algún otro enfoque que se consideraba esencial añadirlo al conjunto. Inevitablemente, estos otros aspectos son los que hacen que generalmente los programadores "se den con la pared" contra ese lenguaje. Por ejemplo, C++ tenía que ser compatible con C (para permitir la migración fácil a los programadores de C), además de eficiente. Estas metas son ambas muy útiles y aportan mucho al éxito de C++, pero también exponen complejidad extra que evita que los proyectos se acaben (ciertamente, se puede echar la culpa a los programadores y la gestión, pero si un lenguaje puede ayudar a capturar los errores, ¿por qué no hacer uso de ello?). Como otro ejemplo, Visual Basic (VB) estaba atado a BASIC, que no estaba diseñado verdaderamente para ser un lenguaje ampliable, por lo que todas las aplicaciones que se apilaban sobre VB producían sintaxis verdaderamente horribles e inmantenibles. Perl es retrocompatible con Awk, Sed, Grep y otras herramientas Unix a las que iba a reemplazar, y como resultado se le acusa a menudo, de producir "código de sólo escritura" (es decir, código que tras unos pocos meses no hay quien lea). Por otro lado, C++, VB, Perl, y otros lenguajes como Smalltalk han visto cómo algunos de sus esfuerzos de diseño se centraban en el aspecto de la complejidad y como resultado son remarcadamente exitosos para solucionar ciertos tipos de problemas.

Lo que más me impresionó es que he llegado a entender que Java parece tener el objetivo de reducir la complejidad *para el programador*. Como si se dijera "no nos importa nada más que reducir el tiempo y la dificultad para producir un código robusto". En los primeros tiempos, esta meta llevaba a un código que no se ejecutaba muy rápido (aunque se habían hecho promesas sobre lo rápido que se ejecutaría Java algún día), pero sin duda ha producido reducciones sorprendentes de tiempo de desarrollo; la mitad o menos del tiempo que lleva crear un programa C++ equivalente. Este resultado sólo puede ahorrar cantidades increíbles de tiempo y dinero, pero Java no se detiene ahí. Envuelve todas las tareas complejas que se han convertido en importantes, como el multihilo y la programación en red, en bibliotecas o aspectos del lenguaje que en ocasiones pueden convertir esas tareas en triviales. Y finalmente, asume muchos problemas de complejidad grandes: programas multiplataforma, cambios dinámicos de código, e incluso seguridad, cada uno de los cuales pueden encajar dentro de un espectro de complejidades que oscila en el rango de "impedimento" a "motivos de cancelación". Por tanto, a pesar de los problemas de rendimiento que se han visto, la promesa de Java es tremenda: puede convertirnos en programadores significativamente más productivos.

Uno de los sitios en los que veo el mayor impacto de esto es en la Web. La programación en red siempre ha sido complicada, y Java la convierte en fácil (y los diseñadores el lenguaje Java están

[1] Esto lo retomo de la 2.ª edición: creo que el lenguaje Python se acerca aún más a esto. Ver http://www.Python.org.

trabajando en facilitarla aún más). La programación en red es como hablar simultáneamente de forma efectiva y de forma más barata de lo que nunca se logró con teléfonos (sólo el correo electrónico ya ha revolucionado muchos negocios). Al intercomunicarnos más, empiezan a pasar cosas divertidas, probablemente mucho más interesantes que las que pasarán con la ingeniería genética.

De todas formas —al crear los programas, trabajar para crear programas, construir interfaces para los programas, de forma que éstos se puedan comunicar con el usuario, ejecutar los programas en distintos tipos de máquinas, y escribir de forma sencilla programas que pueden comunicarse a través de Internet— Java incrementa el ancho de banda de comunicación *entre la gente*. Creo que quizás los resultados de la revolución de la comunicación no se contemplarán por lo que conlleva el transporte de grandes cantidades de bits; veremos la auténtica revolución porque podremos comunicarnos con mayor facilidad: de uno en uno, pero también en grupos y, como planeta. He oído la sugerencia de que la próxima revolución es la formación de cierto tipo de mente global para suficiente gente y suficiente nivel de interconectividad. Puede decirse que Java puede fomentar o no esa revolución, pero al menos la mera posibilidad me ha hecho sentir como si estuviera haciendo algo lleno de sentido al intentar enseñar ese lenguaje.

Prólogo a la 2.ª edición

La gente ha hecho muchos, muchos comentarios maravillosos sobre la primera edición de este libro, cosa que ha sido para mí muy, pero que muy, placentero. Sin embargo, en todo momento habrá quien tenga quejas, y por alguna razón una queja que suele aparecer periódicamente es que "el libro es demasiado grande". Para mí, esto no es verdaderamente una queja, si se reduce a que "tiene demasiadas páginas". (Uno se acuerda de las quejas del Emperador de Austria sobre el trabajo de Mozart: "¡Demasiadas páginas!", y no es que me esté intentando comparar con Mozart de ninguna forma). Además, sólo puedo asumir que semejante queja puede provenir de gente que no tiene aún una idea clara de la vasta extensión del propio lenguaje Java en sí, y que no ha visto el resto de libros sobre la materia —por ejemplo, mi referencia favorita es el *Core Java* de Cay Horstmann & Cary Cornell (Prentice-Hall), que creció tanto que hubo que dividirlo en dos tomos. A pesar de esto, una de las cosas que he intentado hacer en esta edición es eliminar las portes que se han vuelto obsoletas o al menos no esenciales. Me siento a gusto haciendo esto porque el material original sigue en la Web y en el CD ROM que acompaña al libro, en la misma forma de descarga gratuita que la primera edición del libro (en http://www.BruceEckel.com). Si se desea el material antiguo, sigue ahí, y esto es algo maravilloso para un autor. Por ejemplo, puede verse que el último capítulo original, "Proyectos", ya no está aquí; dos de los proyectos se han integrado en los otros capítulos, y el resto ya no son adecuados. También el capítulo de "Patrones de diseño" se volvió demasiado extenso y ha sido trasladado a un libro que versa sobre ellos (descargable también en el sitio web). Por tanto, el libro debería ser más fino.

Pero no lo es.

El aspecto mayor es el continuo desarrollo del lenguaje Java en sí, y en particular las API que se expanden, y prometen proporcionar interfaces estándar para casi todo lo que se desee hacer (y no me sorprendería ver aparecer la API "JTostadora"). Cubrir todas estas API se escapa por supuesto del ámbito de este libro, y es una tarea relegada a otros autores, pero algunos aspectos no pueden ig-

norarse. El mayor de éstos incluye el Java de lado servidor (principalmente Servlets & Java Server Pages o *JSP*), que es verdaderamente una solución excelente al problema de la World Wide Web, donde se descubrió que las distintas plataformas de navegadores web no son lo suficientemente consistentes como para soportar programación en el lado cliente. Además, está todo el problema de crear de forma sencilla aplicaciones que interactúen de forma sencilla con bases de datos, transacciones, seguridad y semejante, cubiertos gracias a los Enterprise Java Beans (EJB). Estos temas están desarrollados en el capítulo que antes se llamaba "Programación de red" y ahora "Computación distribuida", un tema que se está convirtiendo en esencial para todo el mundo. También se verá que se ha compilado este capítulo para incluir un repaso de Jini (pronunciado "yeni", y que no es un acrónimo, sino sólo un nombre), que es una tecnología emergente que permite que cambiemos la forma de pensar sobre las aplicaciones interconectadas. Y por supuesto, el libro se ha cambiado para usar la biblioteca IGU Swing a lo largo de todo el mismo. De nuevo, si se desea el material Java 1.0/1.1 antiguo, es posible conseguirlo gratuitamente del libro de descarga gratuita de *http://www.BruceEckel.com* (también está incluido en el nuevo CD ROM de esta edición, que se adjunta al mismo; hablaré más de él un poco más adelante).

Aparte de nuevas características del lenguaje añadidas a Java 2, y varias correcciones hechas a lo largo de todo el libro, el otro cambio principal está en el capítulo de colecciones que ahora se centra en las colecciones de Java 2, que se usan a lo largo de todo el libro. También he mejorado ese capítulo para que entre más en profundidad en algunos aspectos importantes de las colecciones, en particular, en cómo funcionan las funciones de ***hashing*** (de forma que se puede saber cómo crear una adecuadamente). Ha habido otros movimientos y cambios, incluida la reescritura del Capítulo 1, y la eliminación de algunos apéndices y de otros materiales que ya no consideraba necesarios para el libro impreso, que son un montón de ellos. En general, he intentado recorrer todo, eliminar de la 2.ª edición lo que ya no es necesario (pero que sigue existiendo en la primera edición electrónica), incluir cambios y mejorar todo lo que he podido. A medida que el lenguaje continúa cambiando —aunque no a un ritmo tan frenético como antiguamente— no cabe duda de que habrá más ediciones de este libro.

Para aquellos de vosotros que siguen sin poder soportar el tamaño del libro, pido perdón. Lo creáis o no, he trabajado duro para que se mantenga lo menos posible. A pesar de todo, creo que hay bastantes alternativas que pueden satisfacer a todo el mundo. Además, el libro está disponible electrónicamente (en idioma inglés desde el sitio web, y desde el CD ROM que acompaña al libro), por lo que si se dispone de un ordenador de bolsillo, se puede disponer del libro sin tener que cargar un gran peso. Si sigue interesado en tamaños menores, ya existen de hecho versiones del libro para Palm Pilot. (Alguien me dijo en una ocasión que leería el libro en la cama en su Palm, con la luz encendida a la espalda para no molestar a su mujer. Sólo espero que le ayude a entrar en el mundo de los sueños.) Si se necesita en papel, sé de gente que lo va imprimiendo capítulo a capítulo y se lo lee en el tren.

Java 2

En el momento de escribir el libro, es inminente el lanzamiento del *Java Development Kit* (JDK) 1.3 de Sun, y ya se ha publicado los cambios propuestos para JDK 1.4. Aunque estos números de versión se corresponden aún con los "unos", la forma estándar de referenciar a las versiones posterio-

res a la JDK 1.2 es llamarles "Java 2". Esto indica que hubo cambios muy significativos entre el "viejo Java" —que tenía muchas pegas de las que ya me quejé en la primera edición de este libro— y esta nueva versión más moderna y mejorada del lenguaje, que tiene menos pegas y más adiciones y buenos diseños.

Este libro está escrito para Java 2. Tengo la gran ventaja de librarme de todo el material y escribir sólo para el nuevo lenguaje ya mejorado porque la información vieja sigue existiendo en la 1.ª versión electrónica disponible en la Web y en el CD-ROM (que es a donde se puede ir si se desea obcecarse en el uso de versiones pre-Java 2 del lenguaje). También, y dado que cualquiera puede descargarse gratuitamente el JDK de http://java.sun.com, se supone que por escribir para Java 2, no estoy imponiendo ningún criterio financiero o forzando a nadie a hacer una actualización del software.

Hay, sin embargo, algo que reseñar. JDK 1.3 tiene algunas mejoras que verdaderamente me gustaría usar, pero la versión de Java que está siendo actualmente distribuida para Linux es la JDK 1.2.2 (ver http://www.Linux.org). Linux es un desarrollo importante en conjunción con Java, porque es rápido, robusto, seguro, está bien mantenido y es gratuito; una auténtica revolución en la historia de la computación (no creo que se hayan visto todas estas características unidas en una única herramienta anteriormente). Y Java ha encontrado un nicho muy importante en la programación en el lado servidor en forma de *Servlets*, una tecnología que es una grandísima mejora sobre la programación tradicional basada en CGI (todo ello cubierto en el capítulo "Computación Distribuida").

Por tanto, aunque me gustaría usar sólo las nuevas características, es crítico que todo se compile bajo Linux, y por tanto, cuando se desempaquete el código fuente y se compile bajo ese SO (con el último JDK) se verá que todo compila. Sin embargo, se verá que he puesto notas sobre características de JDK 1.3 en muchos lugares.

El CD ROM

Otro bonus con esta edición es el CD ROM empaquetado al final del libro. En el pasado me he resistido a poner CD ROM al final de mis libros porque pensaba que no estaba justificada una carga de unos pocos Kbytes de código fuente en un soporte tan grande, prefiriendo en su lugar permitir a la gente descargar los elementos desde el sitio web. Sin embargo, pronto se verá que este CD ROM es diferente.

El CD contiene el código fuente del libro, pero también contiene el libro en su integridad, en varios formatos electrónicos. Para mí, el preferido es el formato HTML porque es rápido y está completamente indexado —simplemente se hace clic en una entrada del índice o tabla de contenidos y se estará inmediatamente en esa parte del libro.

La carga de más de 300 Megabytes del CD, sin embargo, es un curso multimedia denominado *Thinking in C: Foundations for C++ & Java*. Originalmente encargué este seminario en CD ROM a Chuck Allison, como un producto independiente, pero decidí incluirlo con la segunda edición tanto de *Thinking in C++* como de *Piensa en Java*, gracias a la consistente experiencia de haber tenido gente viniendo a los seminarios sin la requerida experiencia en C. El pensamiento parece aparentemente ser: "Soy un programador inteligente y no *deseo* aprender C, y sí C++ o Java, por lo que me saltaré C e iré directamente a C++/Java." Tras llegar al seminario, todo el mundo va comprendiendo que el

prerrequisito de aprender C está ahí por algo. Incluyendo el CD ROM con el libro, se puede asegurar que todo el mundo atienda al seminario con la preparación adecuada.

El CD también permite que el libro se presente para una audiencia mayor. Incluso aunque el Capítulo 3 («Controlando el flujo del programa») cubre los aspectos fundamentales de las partes de Java que provienen de C, el CD es una introducción más gentil, y asume incluso un trasfondo de C menor que el que supone el libro. Espero que al introducir el CD será más la gente que se acerque a la programación en Java.

Prólogo a la edición en español

Java se convierte día a día en un *lenguaje de programación universal*; es decir, ya no sólo sirve como lenguaje para programar en entornos de Internet, sino que se está utilizando cada vez más como herramienta de programación orientada a objetos y también como herramienta para cursos específicos de programación o de estructuras de datos, aprovechando sus características de lenguaje *"multiparadigma"*. Por estas razones, los libros que afronten temarios completos y amplios sobre los temas anteriores siempre serán bienvenidos. Si, además de reunir estos requisitos, el autor es uno de los más galardonados por sus obras anteriores, nos enfrentamos ante un reto considerable: "la posibilidad de encontrarnos" ante un gran libro, de esos que hacen "historia". Éste, pensamos, es el caso del libro que tenemos entre las manos. ¿Por qué pensamos así?

El libro como referencia obligada a Java

Piensa en Java introduce todos los fundamentos teóricos y prácticos del lenguaje Java, tratando de explicar con claridad y rigor *no sólo lo que hace* el lenguaje *sino también el porqué.* Eckel introduce los fundamentos de objetos y cómo los utiliza Java. Éste es el caso del estudio que hace de la ocultación de las implementaciones, reutilización de clases y polimorfismo. Además, estudia en profundidad propiedades y características tan importantes como AWT, programación concurrente (multihilo, *multithreading2)*, programación en red, e incluso diseño de patrones.

Es un libro que puede servir para iniciarse en Java y llegar hasta un nivel avanzado. Pero, en realidad se sacará el máximo partido al libro si se conoce otro lenguaje de programación, o al menos técnicas de programación (como haber seguido un curso de Fundamentos de Programación, Metodología de la Programación, Algoritmos, o cursos similares) y ya se puede apostar por un alto y eficiente rendimiento si la migración a Java se hace desde un lenguaje orientado a objetos, como C++.

El libro como formación integral de programador

Una de las fortalezas más notables del libro es su contenido y la gran cantidad de temas importantes cubiertos con toda claridad y rigor, y con gran profundidad. El contenido es muy amplio y sobre todo completo. Eckel prácticamente ha tocado casi todas las técnicas existentes y utilizadas hoy día en el mundo de la programación y de la ingeniería del software. Algunos de los temas más sobresalientes analizados en el libro son: fundamentos de diseño orientado a objetos, implementación de herencia y polimorfismo, manejo de excepciones, multihilo y persistencia, Java en Internet, recolección de basura, paquetes Java, diseño por reutilización: composición, herencia, interfaces y clases internas, arrays y contenedores de clases, clases de E/S Java, programación de redes con *sockets*, JDBC para bases de datos, JSPs *(JavaServer Pages)*, RMI, CORBA, EJBs *(Enterprise JavaBeans)* y Jini, JNI *(Java Native Interface)*.

El excelente y extenso contenido hacen al libro *idóneo* para la preparación de cursos de nivel medio y avanzado de programación, tanto a nivel universitario como profesional. Asimismo, por el enfoque masivamente profesional que el autor da al libro, puede ser una herramienta muy útil como referencia básica o complementaria para preparar los exámenes de certificación Java que la casa Sun Microsystems otorga tras la superación de las correspondientes pruebas. Esta característica es un valor añadido muy importante, al facilitar considerablemente al lector interesado las directrices técnicas necesarias para la preparación de la citada certificación.

Recursos gratuitos en línea

Si las características citadas anteriormente son de por sí lo suficientemente atractivas para la lectura del libro, es sin duda el excelente sitio en Internet del autor otro valor añadido difícil de medir, por no decir inmedible y valiosísimo. La generosidad del autor —y, naturalmente, de Pearson—, que ofrece a cualquier lector, sin necesidad de compra previa, todo el contenido en línea, junto a las frecuentes revisiones de la obra y soluciones a ejercicios seleccionados, con la posibilidad de descargarse gratuitamente todo este inmenso conocimiento incluido en el libro, junto al conocimiento complementario ofertado (ejercicios, revisiones, actualizaciones...), hacen a esta experiencia innovadora del autor digna de los mayores agradecimientos por parte del cuerpo de programadores noveles o profesionales de cualquier lugar del mundo donde se utilice Java (que hoy es prácticamente "todo el mundo mundial", que dirían algunos periodistas).

De igual forma es de agradecer el CD ROM que acompaña al libro y la oferta de un segundo CD gratuito que se puede conseguir siguiendo las instrucciones incluidas en el libro con el texto completo de la versión original en inglés y un gran número de ejercicios seleccionados resueltos y recursos Java de todo tipo.

Para facilitar al lector el uso del CD ROM incluido en el libro, el equipo de revisión técnica ha realizado el Apéndice E: *Correspondencias español-inglés de clases, bases de datos, tablas y campos del CD ROM que acompaña al libro*, a fin de identificar el nombre asignado en la traducción al español, con el nombre original en inglés de dichas clases.

Unas palabras todavía más elogiosas

Para las personas que, como el autor de este prólogo, llevamos muchos años (ya décadas) dedicándonos a programar computadores, enseñar a programar y escribir sobre programación, un libro como éste sólo nos trae elevados y elogiosos pensamientos. Consideramos que es un libro magnífico, maduro, consistente, intelectualmente honesto, bien escrito y preciso. Sin duda, como lo demuestra su larga lista de premios y sus numerosas y magníficas cualidades, *Piensa en Java,* no sólo es una excelente obra para aprender y llegar a dominar el lenguaje Java y su programación, sino también una excelente obra para aprender y dominar las técnicas modernas de programación.

Luis Joyanes Aguilar
Director del Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Universidad Pontificia de Salamanca campus Madrid

Comentarios de los lectores

Mucho mejor que cualquier otro libro de Java que haya visto. Esto se entiende "en orden de magnitud" ... muy completo, con ejemplos directos y al grano, excelentes e inteligentes, sin embarullarse, lleno de explicaciones....En contraste con muchos otros libros de Java lo he encontrado inusualmente maduro, consistente, intelectualmente honesto, bien escrito y preciso. En mi honesta opinión, un libro ideal para estudiar Java. **Anatoly Vorobey, Technion University, Haifa, Israel**.

Uno de los mejores tutoriales de programación, que he visto de cualquier lenguaje. **Joakim Ziegler, FIX sysop**.

Gracias por ese libro maravilloso, maravilloso en Java. **Dr. Gavin Pillary, Registrar, King Eduard VIII Hospital, Suráfrica.**

Gracias de nuevo por este maravilloso libro. Yo estaba completamente perdido (soy un programador que no viene de C) pero tu libro me ha permitido avanzar con la misma velocidad con la que lo he leído. Es verdaderamente fácil entender los principios subyacentes y los conceptos desde el principio, en vez de tener que intentar construir todo el modelo conceptual mediante prueba y error. Afortunadamente podré acudir a su seminario en un futuro no demasiado lejano. **Randall R. Hawley, Automation Technician, Eli Lilly & Co.**

El mejor libro escrito de computadores que haya visto jamás. **Tom Holland**.

Éste es uno de los mejores libros que he leído sobre un lenguaje de programación... El mejor libro sobre Java escrito jamás. **Revindra Pai, Oracle Corporation, línea de productos SUNOS.**

¡Éste es el mejor libro sobre Java que haya visto nunca! Has hecho un gran trabajo. Tu profundidad es sorprendente. Compraré el libro en cuanto se publique. He estado aprendiendo Java desde octubre del 96. He leído unos pocos libros y considero el tuyo uno que "SE DEBE LEER". En estos últimos meses nos hemos centrado en un producto escrito totalmente en Java. Tu libro ha ayudado a consolidar algunos temas en los que andábamos confusos y ha expandido mi conocimiento base. Incluso he usado algunos de tus ejemplos y explicaciones como información en mis entrevistas para ayudar al equipo. He averiguado el conocimiento de Java que tienen preguntándoles por algunas de las cosas que he aprendido a partir de la lectura de tu libro (por ejemplo, la diferencia entre arrays y Vectores). ¡El libro es genial! **Steve Wilkinson, Senior Staff Specialist, MCI Telecommunications**.

Gran libro. El mejor libro de Java que he visto hasta la fecha. **Jeff Sinlair, ingeniero de Software, Kestral Computing**.

Gracias por *Piensa en Java*. Ya era hora de que alguien fuera más allá de una mera descripción del lenguaje para lograr un tutorial completo, penetrante, impactante y que no se centra en los fabricante. He leído casi todos los demás —y sólo el tuyo y el de Patrick Winston han encontrado un lugar en mi corazón. Se lo estoy recomendando ya a los clientes. Gracias de nuevo. **Richard Brooks, consultor de Java, Sun Professional Services, Dallas**.

Otros libros contemplan o abarcan el QUÉ de Java (describiendo la sintaxis y las bibliotecas) o el CÓMO de Java (ejemplos de programación prácticos). *Piensa en Java*[1] es el único libro que conoz-

[1] Thinking in Java (título original de la obra en inglés).

co que explica el PORQUÉ de Java; por qué se diseñó de la manera que se hizo, por qué funciona como lo hace, por qué en ocasiones no funciona, por qué es mejor que C++, por qué no lo es. Aunque hace un buen trabajo de enseñanza sobre el qué y el cómo del lenguaje, *Piensa en Java* es la elección definitiva que toda persona interesada en Java ha de hacer. **Robert S. Stephenson**.

Gracias por escribir un gran libro. Cuanto más lo leo más me gusta. A mis estudiantes también les gusta. **Chuck Iverson**.

Sólo quiero comentarte tu trabajo en *Piensa en Java*. Es la gente como tú la que dignifica el futuro de Internet y simplemente quiero agradecerte el esfuerzo. **Patrick Barrell, Network Officer Mamco, QAF Mgf. Inc.**

La mayoría de libros de Java que existen están bien para empezar, y la mayoría tienen material para principiantes y muchos los mismos ejemplos. El tuyo es sin duda el mejor libro y más avanzado para pensar que he visto nunca. ¡Por favor, publícalo rápido!... También compré *Thinking in C++* simplemente por lo impresionado que me dejó *Piensa en Java*. **George Laframboise, LightWorx Technology Consulting Inc.**

Te escribí anteriormente con mis impresiones favorables relativas a *Piensa en Java* (un libro que empieza prominentemente donde hay que empezar). Y hoy que he podido meterme con Java con tu libro electrónico en mi mano virtual, debo decir (en mi mejor Chevy Chase de *Modern Problems*) "¡Me gusta!". Muy informativo y explicativo, sin que parezca que se lee un texto sin sustancia. Cubres los aspectos más importantes y menos tratados del desarrollo de Java: los porqués. **Sean Brady**.

Tus ejemplos son claros y fáciles de entender. Tuviste cuidado con la mayoría de detalles importantes de Java que no pueden encontrarse fácilmente en la débil documentación de Java. Y no malgastas el tiempo del lector con los hechos básicos que todo programador conoce. **Kai Engert, Innovative Software, Alemania**.

Soy un gran *fan* de *Piensa en Java* y lo he recomendado a mis asociados. A medida que avanzo por la versión electrónica de tu libro de Java, estoy averiguando que has retenido el mismo alto nivel de escritura. **Peter R. Neuvald**.

Un libro de Java MUY BIEN escrito... Pienso que has hecho un GRAN trabajo con él. Como líder de un grupo de interés especial en Java del área de Chicago, he mencionado de forma favorable tu libro y sitio web muy frecuentemente en mis últimas reuniones. Me gustaría usar *Piensa en Java* como la base de cada reunión mensual del grupo, para poder ir repasando y discutiendo sucesivamente cada capítulo. **Mark Ertes**.

Verdaderamente aprecio tu trabajo, y tu libro es bueno. Lo recomiendo aquí a nuestros usuarios y estudiantes de doctorado. **Hughes Leroy // Irisa-Inria Rennes France, jefe de Computación Científica y Transferencia Industrial.**

De acuerdo, sólo he leído unas 40 páginas de *Piensa en Java*, pero ya he averiguado que es el libro de programación mejor escrito y más claro que haya visto jamás... Yo también soy escritor, por lo que probablemente soy un poco crítico. Tengo *Piensa en Java* encargado y ya no puedo esperar más —soy bastante nuevo en temas de programación y no hago más que enfrentarme a curvas de

aprendizaje en todas partes. Por tanto, esto no es más que un comentario rápido para agradecerte este trabajo tan excelente. Ya me había empezado a quemar de tanto navegar por tanta y tanta prosa de tantos y tantos libros de computadores —incluso muchos que venían con magníficas recomendaciones. Me siento muchísimo mejor ahora. **Glenn Becker, Educational Theatre ssociation**.

Gracias por permitirme disponer de este libro tan maravilloso. Lo he encontrado inmensamente útil en el entendimiento final de lo que he experimentado —algo confuso anteriormente— con Java y C++. Leer tu libro ha sido muy gratificante. **Felix Bizaoui, Twin Oaks Industries, Luisa, Va.**

Debo felicitarte por tu excelente libro. He decidido echar un vistazo a *Piensa en Java* guiado por mi experiencia en *Thinking in C++*, y no me ha defraudado. **Jaco van der Merwe, Software Specialist, DataFusion Systems Ltd., Stellenbosch, Suráfrica.**

Este libro hace que todos los demás libros de Java que he leído parezcan un insulto o sin duda inútiles. **Brett g Porter, Senior Programmer, Art & Logic.**

He estado leyendo tu libro durante una semana o dos y lo he comparado con otros libros de Java que he leído anteriormente. Tu libro parece tener un gran comienzo. He recomendado este libro a muchos de mis amigos y todos ellos lo han calificado de excelente. Por favor, acepta mis felicitaciones por escribir un libro tan excelente. **Rama Krishna Bhupathi, Ingeniera de Software, TCSI Corporation, San José**.

Simplemente quería decir lo "brillante" que es tu libro. Lo he estado usando como referencia principal durante mi trabajo de Java hecho en casa. He visto que la tabla de contenidos es justo la más adecuada para localizar rápidamente la sección que se requiere en cada momento. También es genial ver un libro que no es simplemente una compilación de las API o que no trata al programador como a un monigote. **Grant Sayer, Java Components Group Leader, Ceedata Systems Pty Ltd., Australia.**

¡Guau! Un libro de Java profundo y legible. Hay muchos libros pobres (y debo admitir también que un par de ellos buenos) de Java en el mercado, pero por lo que he visto, el tuyo es sin duda uno de los mejores. **John Root, desarrollador Web, Departamento de la Seguridad Social, Londres.**

Acabo de empezar *Piensa en Java*. Espero que sea bueno porque me gustó mucho *Thinking in C++* (que leí como programador ya experimentado en C++, intentado adelantarme a la curva de aprendizaje). En parte estoy menos habituado a Java, pero espero que el libro me satisfaga igualmente. Eres un autor maravilloso. **Kevin K. Lewis, Tecnólogo, ObjectSpace Inc.**

Creo que es un gran libro. He aprendido todo lo que sé de Java a partir de él. Gracias por hacerlo disponible gratuitamente a través de Internet. Si no lo hubieras hecho no sabría nada de Java. Pero lo mejor es que tu libro no es un mero folleto publicitario de Java. También muestra sus lados negativos. TÚ has hecho aquí un gran trabajo. **FrederikFix, Bélgica**.

Siempre me han enganchado tus libros. Hace un par de años, cuando quería empezar con C++, fue *C++ Inside & Out* el que me introdujo en el fascinante mundo de C++. Me ayudó a disponer de mejores oportunidades en la vida. Ahora, persiguiendo más conocimiento y cuando quería aprender Java, me introduje en *Piensa en Java* —sin dudar de que gracias a él ya no necesitaría ningún otro libro. Simplemente fantástico. Es casi como volver a descubrirme a mí mismo a medida que avanzo

en el libro. Apenas hace un mes que he empezado con Java y mi corazón late gracias a ti. Ahora lo entiendo todo mucho mejor. **Anand Kumar S., ingeniero de Software Computervision, India.**

Tu libro es una introducción general excelente. **Peter Robinson, Universidad de Cambridge, Computar Laboratory.**

Es con mucho el mejor material al que he tenido acceso al aprender Java y simplemente quería que supieras la suerte que he tenido de poder encontrarlo. ¡GRACIAS! **Chuck Peterson, Product Leader, Internet Product Line, IVIS International.**

Este libro es genial. Es el tercer libro de Java que he empezado y ya he recorrido prácticamente dos tercios. Espero acabar éste. Me he enterado de su existencia porque se usa en algunas clases internas de Lucen Technologies y un amigo me ha dicho que el libro estaba en la Red. Buen trabajo. **Jerry Nowlin, MTS, Lucent Technologies.**

De los aproximadamente seis libros de Java que he acumulado hasta la fecha, tu *Piensa en Java* es sin duda el mejor y el más claro. **Michael Van Waas, doctor, presidente, TMR Associates.**

Simplemente quiero darte las gracias por *Piensa en Java*. ¡Qué libro tan maravilloso has hecho! ¡Y para qué mencionar el poder bajárselo gratis! Como estudiante creo que tus libros son de valor incalculable, tengo una copia de *C++ Inside & Out*, otro gran libro sobre C++), porque no sólo me enseñan el cómo hacerlo, sino que también los porqués, que sin duda son muy importantes a la hora de sentar unas buenas bases en lenguajes como C++ y Java. Tengo aquí bastantes amigos a los que les encanta programar como a mí, y les he hablado de tus libros. ¡Todos piensan que son geniales! Por cierto, soy indonesio y vivo en Java. **Ray Frederick Djajadinata, estudiante en Trisakti University, Jakarta.**

El mero hecho de que hayas hecho que este trabajo esté disponible gratuitamente en la Red me deja conmocionado. Pensé que debía decirte cuánto aprecio y respeto lo que estás haciendo. **Shane Ke-Bouthillier, estudiante de Ingeniería en Informática, Universidad de Alberta, Canadá.**

Tengo que decirte cuánto ansío leer tu columna mensual. Como novato en el mundo de la programación orientada a objetos, aprecio el tiempo y el grado de conocimiento que aportas en casi todos los temas elementales. He descargado tu libro, pero puedes apostar a que compraré una copia en papel en cuanto se publique. Gracias por toda tu ayuda. **Dan Cashmer, D. C. Ziegler & Co.**

Simplemente quería felicitarte por el buen trabajo que has hecho. Primero me recorrí la versión PDF de *Piensa en Java*. Incluso antes de acabar de leerla, corrí a la tienda y compré *Thinking in C++*. Ahora que llevo en el negocio de la informática ocho años, como consultor, ingeniero de software, profesor/formador, y últimamente autónomo, creo que puedo decir que he visto suficiente (fíjate que no digo haber visto "todo" sino suficiente). Sin embargo, estos libros hacen que mi novia me llame "geek". No es que tenga nada contra el concepto en sí —simplemente pensaba que ya había dejado atrás esta fase. Pero me veo a mí mismo disfrutando sinceramente de ambos libros, de una forma que no había sentido con ningún otro libro que haya tocado o comprado hasta la fecha. Un estilo de escritura excelente, una introducción genial de todos los temas y mucha sabiduría en ambos textos. Bien hecho. **Simon Goland, simonsez@smartt.com, Simon Says Consulting, Inc.**

¡Debo decir que tu *Piensa en Java* es genial! Es exactamente el tipo de documentación que buscaba. Especialmente las secciones sobre los buenos y malos diseños basados en Java. **Dirk Duehr, Lexikon Verlag, Bertelsmann AG, Alemania.**

Gracias por escribir dos grandes libros (*Thinking in C++, Piensa en Java*). Me has ayudado inmensamente en mi progresión en la programación orientada a objetos. **Donald Lawon, DCL Enterprises.**

Gracias por tomarte el tiempo de escribir un libro de Java que ayuda verdaderamente. Si enseñar hace que aprendas algo, tú ya debes estar más que satisfecho. **Dominic Turner, GEAC Support.**

Es el mejor libro de Java que he leído jamás —y he leído varios. **Jean-Yves MENGANT, Chief Software Architect NAT-SYSTEM, París, Francia.**

Piensa en Java proporciona la mejor cobertura y explicación. Muy fácil de leer, y quiero decir que esto se extiende también a los fragmentos de código. **Ron Chan, Ph. D., Expert Choice Ind., Pittsburg PA.**

Tu libro es genial. He leído muchos libros de programación y el tuyo sigue añadiendo luz a la programación en mi mente. **Ningjian Wang, Information System Engineer, The Vanguard Group.**

Piensa en Java es un libro excelente y legible. Se lo recomiendo a todos mis alumnos. **Dr. Paul Gorman, Department of Computer Sciente, Universidad de Otago, Dunedin, Nueva Zelanda.**

Haces posible que exista el proverbial almuerzo gratuito, y no simplemente una comida basada en sopa de pollo, sino una delicia de *gourmet* para aquéllos que aprecian el buen software y los libros sobre él mismo. **Jose Suriol, Scylax Corporation.**

¡Gracias por la oportunidad de ver cómo este libro se convierte en una obra maestra! ES EL MEJOR libro de la materia que he leído o recorrido. **Jeff Lapchinsky, programador, Net Result Tecnologies.**

Tu libro es conciso, accesible y gozoso de leer. **Keith Ritchie, Java Research & Develpment Team, KL Group Inc.**

¡Es sin duda el mejor libro de Java que he leído! **Daniel Eng.**

¡Es el mejor libro de Java que he visto! **Rich Hoffarth, Arquitecto Senior, West Group.**

Gracias por un libro tan magnífico. Estoy disfrutando mucho a medida que leo capítulos. **Fred Trimble, Actium Corporation**.

Has llegado a la maestría en el arte de hacernos ver los detalles, despacio y con éxito. Haces que la lectura sea MUY fácil y satisfactoria. Gracias por un tutorial tan verdaderamente maravilloso. **Rajesh Rau, Software Consultant.**

Piensa en Java es un rock para el mundo libre! **Miko O'Sullivan, Presidente, Idocs Inc.**

Sobre Thinking in C++:

Best Book! Ganador en 1995 del Software Development Magazine Jolt Award!

"Este libro es un tremendo logro. Deberías tener una copia en la estantería. El capítulo sobre flujos de E/S presenta el tratamiento más comprensible y fácil de entender sobre ese tema que jamás haya visto."

Al Stevens
Editor, *Doctor Dobbs Journal*

"El libro de Eckel es el único que explica claramente cómo replantearse la construcción de programas para la orientación a objetos. Que el libro es también un tutorial excelente en las entradas y en las salidas de C++ es un valor añadido."

Andrew Binstock
Editor, *Unix Review*

"Bruce continúa deleitándome con esta introspección de C++, y *Thinking in C++* es la mejor colección de ideas hasta la fecha. Si se desean respuestas rápidas a preguntas difíciles sobre C++, compre este libro tan sobresaliente."

Gary Entsminger
Autor, *The Tao of Objects*

"*Thinking in C++*" explora paciente y metódicamente los aspectos de cuándo y cómo usar los interlineados, referencias, sobrecargas de operadores, herencia, y objetos dinámicos, además de temas avanzados como el uso adecuado de plantillas, excepciones y la herencia múltiple. Todo el esfuerzo se centra en un producto que engloba la propia filosofía de Eckel del diseño de objetos y programas. Un libro que no debe faltar en la librería de un desarrollador de C++, *Piensa en Java* es el libro de C++ que hay que tener si se están haciendo desarrollos serios con C++."

Richard Hale Shaw
Ayudante del Editor, *PC Magazine*

Introducción

Como cualquier lenguaje humano, Java proporciona una forma de expresar conceptos. Si tiene éxito, la expresión media será significativamente más sencilla y más flexible que las alternativas, a medida que los problemas crecen en tamaño y complejidad.

No podemos ver Java como una simple colección de características —algunas de las características no tienen sentido aisladas. Se puede usar la suma de partes sólo si se está pensando en *diseño*, y no simplemente en codificación. Y para entender Java así, hay que entender los problemas del lenguaje y de la programación en general. Este libro habla acerca de problemas de programación, por qué son problemas y el enfoque que Java sigue para solucionarlos. Por consiguiente, algunas características que explico en cada capítulo se basan en cómo yo veo que se ha solucionado algún problema en particular con el lenguaje. Así, espero conducir poco a poco al lector, hasta el punto en que Java se convierta en lengua casi materna.

Durante todo el tiempo, estaré tomando la actitud de que el lector construya un modelo mental que le permita desarrollar un entendimiento profundo del lenguaje; si se encuentra un puzzle se podrá alimentar de éste al modelo para tratar de deducir la respuesta.

Prerrequisitos

Este libro asume que se tiene algo de familiaridad con la programación: se entiende que un programa es una colección de sentencias, la idea de una subrutina/función/macro, sentencias de control como "if" y bucles estilo "while", etc. Sin embargo, se podría haber aprendido esto en muchos sitios, como, por ejemplo, la programación con un lenguaje de macros o el trabajo con una herramienta como Perl. A medida que se programa hasta el punto en que uno se siente cómodo con las ideas básicas de programación, se podrá ir trabajando a través de este libro. Por supuesto, el libro será más fácil para los programadores de C y aún más para los de C++, pero tampoco hay por qué excluirse a sí mismo cuando se desconocen estos lenguajes (aunque en este caso es necesario tener la voluntad de trabajar duro; además, el CD multimedia que acompaña a este texto te permitirá conocer rápidamente los conceptos de la sintaxis de C necesarios para aprender Java). Presentaré los conceptos de la programación orientada a objetos (POO) y los mecanismos de control básicos de Java, para tener conocimiento de ellos, y los primeros ejercicios implicarán las secuencias de flujo de control básicas.

Aunque a menudo aparecerán referencias a aspectos de los lenguajes C y C++, no deben tomarse como comentarios profundos, sino que tratan de ayudar a los programadores a poner Java en perspectiva con esos lenguajes, de los que, después de todo, es de los que desciende Java. Intentaré hacer que estas referencias sean lo más simples posibles, y explicar cualquier cosa que crea que una persona que no haya programado nunca en C o C++ pueda desconocer.

Aprendiendo Java

Casi a la vez que mi primer libro *Using C++* (Osborne/McGraw-Hill, 1989)apareció, empecé a enseñar ese lenguaje. Enseñar lenguajes de programación se ha convertido en mi profesión; he visto cabezas dudosas, caras en blanco y expresiones de puzzle en audiencias de todo el mundo desde 1989. A medida que empecé con formación *in situ* a grupos de gente más pequeños, descubrí algo en los ejercicios. Incluso aquéllos que sonreían tenían pegas con muchos aspectos. Al dirigir la sesión de C++ en la *Software Development Conference* durante muchos años (y después la sesión de Java), descubrí que tanto yo como otros oradores tendíamos a ofrecer a la audiencia, en general, muchos temas demasiado rápido. Por tanto, a través, tanto de la variedad del nivel de audiencia como de la forma de presentar el material, siempre se acababa perdiendo parte de la audiencia. Quizás es pedir demasiado, pero dado que soy uno de ésos que se resisten a las conferencias tradicionales (y en la mayoría de casos, creo que esta resistencia proviene del aburrimiento), quería intentar algo que permitiera tener a todo el mundo enganchado.

Durante algún tiempo, creé varias presentaciones diferentes en poco tiempo. Por consiguiente, acabé aprendiendo a base de experimentación e iteración (una técnica que también funciona bien en un diseño de un programa en Java). Eventualmente, desarrollé un curso usando todo lo que había aprendido de mi experiencia en la enseñanza —algo que me gustaría hacer durante bastante tiempo. Descompone el problema de aprendizaje en pasos discretos, fáciles de digerir, y en un seminario en máquina (la situación ideal de aprendizaje) hay ejercicios seguidos cada uno de pequeñas lecciones. Ahora doy cursos en seminarios públicos de Java, que pueden encontrarse en http://www.BruceEckel.com. (El seminario introductorio también está disponible como un CD ROM. En el sitio web se puede encontrar más información al respecto.)

La respuesta que voy obteniendo de cada seminario me ayuda a cambiar y reenfocar el material hasta que creo que funciona bien como medio docente. Pero este libro no es simplemente un conjunto de notas de los seminarios —intenté empaquetar tanta información como pude en este conjunto de páginas, estructurándola de forma que cada tema te vaya conduciendo al siguiente. Más que otra cosa, el libro está diseñado para servir al lector solitario que se está enfrentando y dando golpes con un nuevo lenguaje de programación.

Objetivos

Como en mi libro anterior *Thinking in C++*, este libro pretende estar estructurado en torno al proceso de enseñanza de un lenguaje. En particular, mi motivación es crear algo que me proporcione una forma de enseñar el lenguaje en mis propios seminarios. Cuando pienso en un capítulo del libro, lo pienso en términos de lo que constituiría una buena lección en un seminario. Mi objetivo es lograr fragmentos que puedan enseñarse en un tiempo razonable, seguidos de ejercicios que sean fáciles de llevar a cabo en clase.

Mis objetivos en este libro son:

1. Presentar el material paso a paso de forma que se pueda digerir fácilmente cada concepto antes de avanzar.

2. Utilizar ejemplos que sean tan simples y cortos como se pueda. Esto evita en ocasiones acometer problemas del "mundo real", pero he descubierto que los principiantes suelen estar más contentos cuando pueden entender todos los detalles de un ejemplo que cuando se ven impresionados por el gran rango del problema que solucionan. Además, hay una limitación severa de cara a la cantidad de código que se puede absorber en una clase. Por ello, no dudaré en recibir críticas por usar "ejemplos de juguete", sino que estoy deseoso de aceptarlas en aras de lograr algo pedagógicamente útil.

3. Secuenciar cuidadosamente la presentación de características de forma que no se esté viendo algo que aún no se ha expuesto. Por supuesto, esto no es siempre posible; en esas situaciones se dan breves descripciones introductorias.

4. Dar lo que yo considero que es importante que se entienda del lenguaje, en lugar de todo lo que sé. Creo que hay una jerarquía de importancia de la información, y que hay hechos que el 95% de los programadores nunca necesitarán saber y que simplemente confunden a la gente y añaden su percepción de la complejidad del lenguaje. Por tomar un ejemplo de C, si se memoriza la tabla de precedencia de los operadores (algo que yo nunca hice) se puede escribir un código más inteligente. Pero si se piensa en ello, también confundirá la legibilidad y mantenibilidad de ese código. Por tanto, hay que olvidarse de la precedencia, y usar paréntesis cuando las cosas no estén claras.

5. Mantener cada sección lo suficientemente enfocada de forma que el tiempo de exposición —el tiempo entre periodos de ejercicios— sea pequeño. Esto no sólo mantiene más activas las mentes de la audiencia, que están en un seminario en máquina, sino que también transmite más sensación de avanzar.

6. Proporcionar una base sólida que permita entender los aspectos lo suficientemente bien como para avanzar a cursos y libros más difíciles.

Documentación en línea

El lenguaje Java y las bibliotecas de Sun Microsystems (de descarga gratuita) vienen con su documentación en forma electrónica, legible utilizando un navegador web, y casi toda implementación de Java de un tercero tiene éste u otro sistema de documentación equivalente. Casi todos los libros publicados de Java, incorporan esta documentación. Por tanto, o ya se tiene, o se puede descargar, y a menos que sea necesario, este libro no repetirá esa documentación pues es más rápido encontrar las descripciones de las clases en el navegador web que buscarlas en un libro (y la documentación en línea estará probablemente más actualizada). Este libro proporcionará alguna descripción extra de las clases sólo cuando sea necesario para complementar la documentación, de forma que se pueda entender algún ejemplo particular.

Capítulos

Este libro se diseñó con una idea en la cabeza: la forma que tiene la gente de aprender Java. La realimentación de la audiencia de mis seminarios me ayudó a ver las partes difíciles que necesitaban aclaraciones. En las áreas en las que me volvía ambiguo e incluía varias características a la vez, descubrí —a través del proceso de presentar el material— que si se incluyen muchas características de golpe, hay que explicarlas todas, y esto suele conducir fácilmente a la confusión por parte del alumno. Como resultado, he tenido bastantes problemas para presentar las características agrupadas de tan pocas en pocas como me ha sido posible.

El objetivo, por tanto, es que cada capítulo enseñe una única característica, o un pequeño grupo de características asociadas, sin pasar a características adicionales. De esa forma se puede digerir cada fragmento en el contexto del conocimiento actual antes de continuar.

He aquí una breve descripción de los capítulos que contiene el libro, que corresponde a las conferencias y periodos de ejercicio en mis seminarios en máquina.

Capítulo 1: Introducción a los objetos

Este capítulo presenta un repaso de lo que es la programación orientada a objetos, incluyendo la respuesta a la cuestión básica "¿Qué es un objeto?", interfaz frente a implementación, abstracción y encapsulación, mensajes y funciones, herencia y composición, y la importancia del polimorfismo. También se obtendrá un repaso a los aspectos de la creación de objetos como los constructores, en los que residen los objetos, dónde ponerlos una vez creados, y el mágico recolector de basura que limpia los objetos cuando dejan de ser necesarios. Se presentarán otros aspectos, incluyendo el manejo de errores con excepciones, el multihilo para interfaces de usuario con buen grado de respuesta, y las redes e Internet. Se aprenderá qué convierte a Java en especial, por qué ha tenido tanto éxito, y también algo sobre análisis y diseño orientado a objetos.

Capítulo 2: Todo es un objeto

Este capítulo te lleva al punto donde tú puedas crear el primer programa en Java, por lo que debe dar un repaso a lo esencial, incluyendo el concepto de *referencia* a un objeto; cómo crear un objeto; una introducción de los tipos primitivos y arrays; el alcance y la forma en que destruye los objetos el recolector de basura; cómo en Java todo es un nuevo tipo de datos (clase) y cómo crear cada uno sus propias clases; funciones, argumentos y valores de retorno; visibilidad de nombres y el uso de componentes de otras bibliotecas; la palabra clave **static**; y los comentarios y documentación embebida.

Capítulo 3: Controlando el flujo de los programas

Este capítulo comienza con todos los operadores que provienen de C y C++. Además, se descubrirán los fallos de los operadores comunes, la conversión de tipos, la promoción y la precedencia. Des-

pués se presentan las operaciones básicas de control de flujo y selección existentes en casi todos los lenguajes de programación: la opción con *if-else*; los bucles con *while* y *for*; cómo salir de un bucle con *break* y *continue*, además de sus versiones etiquetadas en Java (que vienen a sustituir al "goto perdido" en Java); la selección con *switch*. Aunque gran parte de este material tiene puntos comunes con el código de C y C++, hay algunas diferencias. Además, todos los ejemplos estarán hechos completamente en Java por lo que el lector podrá estar más a gusto con la apariencia de Java.

Capítulo 4: Inicialización y limpieza

Este capítulo comienza presentando el constructor, que garantiza una inicialización adecuada. La definición de constructor conduce al concepto de sobrecarga de funciones (puesto que puede haber varios constructores). Éste viene seguido de una discusión del proceso de limpieza, que no siempre es tan simple como parece. Normalmente, simplemente se desecha un objeto cuando se ha acabado con él y el recolector de basura suele aparecer para liberar la memoria. Este apartado explora el recolector de basura y algunas de sus idiosincrasias. El capítulo concluye con un vistazo más cercano a cómo se inicializan las cosas: inicialización automática de miembros, especificación de inicialización de miembros, el orden de inicialización, la inicialización **static** y la inicialización de arrays.

Capítulo 5: Ocultando la implementación

Este capítulo cubre la forma de empaquetar junto el código, y por qué algunas partes de una biblioteca están expuestas a la vez que otras partes están ocultas. Comienza repasando las palabras clave **package** e **import**, que llevan a cabo empaquetado a nivel de archivo y permiten construir bibliotecas de clases. Después examina el tema de las rutas de directorios y nombres de fichero. El resto del capítulo echa un vistazo a las palabras clave **public**, **private** y **protected**, el concepto de acceso "friendly", y qué significan los distintos niveles de control de acceso cuando se usan en los distintos conceptos.

Capítulo 6: Reutilizando clases

El concepto de herencia es estándar en casi todos los lenguajes de POO. Es una forma de tomar una clase existente y añadirla a su funcionalidad (además de cambiarla, que será tema del Capítulo 7). La herencia es a menudo una forma de reutilizar código dejando igual la "clase base", y simplemente parcheando los elementos aquí y allí hasta obtener lo deseado. Sin embargo, la herencia no es la única forma de construir clases nuevas a partir de las existentes. También se puede empotrar un objeto dentro de una clase nueva con la *composición*. En este capítulo, se aprenderán estas dos formas de reutilizar código en Java, y cómo aplicarlas.

Capítulo 7: Polimorfismo

Cada uno por su cuenta, podría invertir varios meses para descubrir y entender el polimorfismo, claves en POO. A través de pequeños ejemplos simples, se verá cómo crear una familia de tipos con

herencia y manipular objetos de esa familia a través de su clase base común. El polimorfismo de Java permite tratar los objetos de una misma familia de forma genérica, lo que significa que la mayoría del código no tiene por qué depender de un tipo de información específico. Esto hace que los programas sean extensibles, por lo que se facilita y simplifica la construcción de programas y el mantenimiento de código.

Capítulo 8: Interfaces y clases internas

Java proporciona una tercera forma de establecer una relación de reutilización a través de la *interfaz*, que es una abstracción pura del interfaz de un objeto. La **interfaz** es más que una simple clase abstracta llevada al extremo, puesto que te permite hacer variaciones de la "herencia múltiple" de C++, creando una clase sobre la que se puede hacer una conversión hacia arriba a más de una clase base.

A primera vista, las clases parecen un simple mecanismo de ocultación de código: se colocan clases dentro de otras clases. Se aprenderá, sin embargo, que la clase interna hace más que eso —conoce y puede comunicarse con la clase contenedora— y que el tipo de código que se puede escribir con clases internas es más elegante y limpio, aunque es un concepto nuevo para la mayoría de la gente y lleva tiempo llegar a estar cómodo utilizando el diseño clases internas.

Capítulo 9: Guardando tus objetos

Es un programa bastante simple que sólo tiene una cantidad fija de objetos de tiempo de vida conocido. En general, todos los programas irán creando objetos nuevos en distintos momentos, conocidos sólo cuando se está ejecutando el programa. Además, no se sabrá hasta tiempo de ejecución la cantidad o incluso el tipo exacto de objetos que se necesitan. Para solucionar el problema de programación general, es necesario crear cualquier número de objetos, en cualquier momento y en cualquier lugar. Este capítulo explora en profundidad la biblioteca de contenedores que proporciona Java 2 para almacenar objetos mientras se está trabajando con ellos: los simples arrays y contenedores más sofisticados (estructuras de datos) como **ArrayList** y **HashMap**.

Capítulo 10: Manejo de errores con excepciones

La filosofía básica de Java es que el código mal formado no se ejecutará. En la medida en que sea posible, el compilador detecta problemas, pero en ocasiones los problemas —debidos a errores del programador o a condiciones de error naturales que ocurren como parte de la ejecución normal del programa— pueden detectarse y ser gestionados sólo en tiempo de ejecución. Java tiene el *manejo de excepciones* para tratar todos los problemas que puedan surgir al ejecutar el programa. Este capítulo muestra cómo funcionan en Java las palabras clave **try**, **catch**, **throw**, **throws** y **finally**; cuándo se deberían lanzar excepciones y qué hacer al capturarlas. Además, se verán las excepciones estándar de Java, cómo crear las tuyas propias, qué ocurre con las excepciones en los constructores y cómo se ubican los gestores de excepciones.

Capítulo 11: El sistema de E/S de Java

Teóricamente, se puede dividir cualquier programa en tres partes: entrada, proceso y salida. Esto implica que la E/S (entrada/salida) es una parte importante de la ecuación. En este capítulo se aprenderá las distintas clases que proporciona Java para leer y escribir ficheros, bloques de memoria y la consola. También se mostrará la distinción entre E/S "antigua" y "nueva". Además, este capítulo examina el proceso de tomar un objeto, pasarlo a una secuencia de bytes (de forma que pueda ser ubicado en el disco o enviado a través de una red) y reconstruirlo, lo que realiza automáticamente la *serialización de objetos de Java.* Además, se examinan las bibliotecas de compresión de Java, que se usan en el formato de archivos de Java (JAR).

Capítulo 12: Identificación de tipos en tiempo de ejecución

La identificación de tipos en tiempo de ejecución (RTTI) te permite averiguar el tipo exacto de un objeto cuando se tiene sólo una referencia al tipo base. Normalmente, se deseará ignorar intencionadamente el tipo exacto de un objeto y dejar que sea el mecanismo de asignación dinámico de Java (polimorfismo) el que implemente el comportamiento correcto para ese tipo. A menudo, esta información te permite llevar a cabo operaciones de casos especiales, más eficientemente. Este capítulo explica para qué existe la RTTI, cómo usarlo, y cómo librarse de él cuando sobra. Además, este capítulo presenta el mecanismo de *reflectividad* de Java.

Capítulo 13: Creación de ventanas y applets

Java viene con la biblioteca IGU Swing, que es un conjunto de clases que manejan las ventanas de forma portable. Estos programas con ventanas pueden o bien ser applets o bien aplicaciones independientes. Este capítulo es una introducción a Swing y a la creación de applets de World Wide Web. Se presenta la importante tecnología de los "JavaBeans", fundamental para la creación de herramientas de construcción de programas de Desarrollo Rápido de Aplicaciones (RAD).

Capítulo 14: Hilos múltiples

Java proporciona una utilidad preconstruida para el soporte de múltiples subtareas concurrentes denominadas *hilos*, que se ejecutan en un único programa. (A menos que se disponga de múltiples procesadores en la máquina, los múltiples hilos sólo son *aparentes.*) Aunque éstas pueden usarse en todas partes, los hilos son más lucidos cuando se intenta crear una interfaz de usuario con alto grado de respuesta, de forma que, por ejemplo, no se evita que un usuario pueda presionar un botón o introducir datos mientras se está llevando a cabo algún procesamiento. Este capítulo echa un vistazo a la sintaxis y la semántica del multihilo en Java.

Capítulo 15: Computación distribuida

Todas las características y bibliotecas de Java aparecen realmente cuando se empieza a escribir programas que funcionen en red. Este capítulo explora la comunicación a través de redes e Internet, y las clases que proporciona Java para facilitar esta labor. Presenta los tan importantes conceptos de *Servlets* y *JSP* (para programación en el lado servidor), junto con *Java DataBase Connectivity* (JDBC) y el *Remote Method Invocation* (RMI). Finalmente, hay una introducción a las nuevas tecnologías de *JINI*, *JavaSpaces*, y *Enterprise JavaBeans (EJBS)*.

Apéndice A: Paso y retorno de objetos

Puesto que la única forma de hablar con los objetos en Java es mediante referencias, los conceptos de paso de objetos a una función y de devolución de un objeto de una función tienen algunas consecuencias interesantes. Este apéndice explica lo que es necesario saber para gestionar objetos cuando se está entrando y saliendo de funciones, y también muestra la clase **String**, que usa un enfoque distinto al problema.

Apéndice B: La Interfaz Nativa de Java (JNI)

Un programa Java totalmente portable tiene importantes pegas: la velocidad y la incapacidad para acceder a servicios específicos de la plataforma. Cuando se conoce la plataforma sobre la que está ejecutando, es posible incrementar dramáticamente la velocidad de ciertas operaciones construyéndolas como *métodos nativos*, que son funciones escritas en otro lenguaje de programación (actualmente, sólo están soportados C/C++). Este apéndice da una introducción más que satisfactoria que debería ser capaz de crear ejemplos simples que sirvan de interfaz con código no Java.

Apéndice C: Guías de programación Java

Este apéndice contiene sugerencias para guiarle durante la realización del diseño de programas de bajo nivel y la escritura de código.

Apéndice D: Lecturas recomendadas

Una lista de algunos libros sobre Java que he encontrado particularmente útil.

Ejercicios

He descubierto que los ejercicios simples son excepcionalmente útiles para completar el entendimiento de los estudiantes durante un seminario, por lo que se encontrará un conjunto de ellos al final de cada capítulo.

La mayoría de ejercicios están diseñados para ser lo suficientemente sencillos como para poder ser resueltos en un tiempo razonable en una situación de clase mientras que observa el profesor, asegurándose de que todos los alumnos asimilen el material. Algunos ejercicios son más avanzados para evitar que los alumnos experimentados se aburran. La mayoría están diseñados para ser resueltos en poco tiempo y probar y pulir el conocimiento. Algunos suponen un reto, pero ninguno presenta excesivas dificultades. (Presumiblemente, cada uno podrá encontrarlos —o más probablemente te encontrarán ellos a ti.)

En el documento electrónico *The Thinking in Java Annotated Solution Guide* pueden encontrarse soluciones a ejercicios seleccionados, disponibles por una pequeña tasa en http://www.BruceEckel.com.

CD ROM Multimedia

Hay dos CD multimedia asociados con este libro. El primero está en el propio libro: *Thinking in C*, descritos al final del prefacio, que te preparan para el libro aportando velocidad en la sintaxis de C necesaria para poder entender Java.

Hay disponible un segundo CD ROM multimedia, basado en los contenidos del libro. Este CD ROM es un producto separado y contiene los contenidos **enteros** del seminario de formación "Hands-On Java" de una semana de duración. Esto son grabaciones de conferencias de más de 15 horas que he grabado, y sincronizado con cientos de diapositivas de información. Dado que el seminario se basa en este libro, es el acompañamiento ideal.

El CD ROM contiene todas las conferencias (¡con la importante excepción de la atención personalizada!) de los seminarios de cinco días de inmersión total. Creemos que establece un nuevo estándar de calidad.

El CD ROM "Hands-On Java" está disponible sólo bajo pedido, que se cursa directamente del sitio web *http://www.BruceEckel.com*.

Código fuente

Todo el código fuente de este libro está disponible de modo gratuito sometido a *copyright*, distribuido como un paquete único, visitando el sitio web *http://www.BruceEckel.com*. Para asegurarse de obtener la versión más actual, éste es el lugar oficial para distribución del código y de la versión electrónica del libro. Se pueden encontrar versiones espejo del código y del libro en otros sitios (algunos de éstos están referenciados en *http://www.BruceEckel.com)*, pero habría que comprobar el sitio oficial para asegurarse de obtener la edición más reciente. El código puede distribuirse en clases y en otras situaciones con fines educativos.

La meta principal del *copyright* es asegurar que el código fuente se cite adecuadamente, y prevenir que el código se vuelva a publicar en medios impresos sin permiso. (Mientras se cite la fuente, utilizando los ejemplos del libro, no habrá problema en la mayoría de los medios.)

En cada fichero de código fuente, se encontrará una referencia a la siguiente nota de *copyright*:

```
//:! :CopyRght.txt
Copyright (c)2000 Bruce Eckel
Source code file from the 2nd edition of the book
"Thinking in Java." All rights reserved EXCEPT as
allowed by the following statements:
You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in Java" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
Bruce Eckel or the publisher be liable for any
lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///:~
```

El código puede usarse en proyectos y en clases (incluyendo materiales de presentación) mientras se mantenga la observación de *copyright* que aparece en cada archivo fuente.

Estándares de codificación

En el texto de este libro, los identificadores (nombres de funciones, variables y clases) están en **negrita**. La mayoría de palabras clave también están en negrita, excepto en aquellos casos en que las palabras se usan tanto que ponerlas en negrita podría volverse tedioso, como es el caso de la palabra "clase".

Para los ejemplos de este libro, uso un estilo de codificación bastante particular. Este estilo sigue al estilo que la propia Sun usa en prácticamente todo el código de sitio web (véase *http://java.sun.com/docs/codeconv/index.html*), y parece que está soportado por la mayoría de entornos de desarrollo Java. Si ha leído el resto de mis trabajos, también verá que el estilo de codificación de Sun coincide con el mío —esto me alegra, aunque no tenía nada que hacer con él. El aspecto del estilo de formato es bueno para lograr horas de tenso debate, por lo que simplemente diré que no pretendo dictar un estilo correcto mediante mis ejemplos; tengo mi propia motivación para usar el estilo que uso. Java es un lenguaje de programación de forma libre, se puede seguir usando cualquier estilo con el que uno esté a gusto.

Los programas de este libro son archivos incluidos por el procesador de textos, directamente sacados de archivos compilados. Por tanto, los archivos de código impresos en este libro deberían funcionar sin errores de compilador. Los errores que *deberían* causar mensajes de error en tiempo de compilación están comentados o marcados mediante //!, por lo que pueden ser descubiertos fácilmente, y probados utilizando medios automáticos. Los errores descubiertos de los que ya se haya informado al autor, aparecerán primero en el código fuente distribuido y posteriormente en actualizaciones del libro (que también aparecerán en el sitio web *http://www.BruceEckel.com*).

Versiones de Java

Generalmente confío en la implementación que Sun hace de Java como referencia para definir si un determinado comportamiento es o no correcto.

Con el tiempo, Sun ha lanzado tres versiones principales de Java: la 1.0, la 1.1 y la 2 (que se llama versión 2, incluso aunque las versiones del JDK de Sun siguen usando el esquema de numeración de 1.2, 1.3, 1.4, etc.). La versión 2 parece llevar finalmente a Java a la gloria, especialmente en lo que concierne a las herramientas de interfaces. Este libro se centra en, y está probado con, Java 2, aunque en ocasiones hago concesiones a las características anteriores de Java 2, de forma que el código pueda compilarse bajo Linux (vía el JDK de Linux que estaba disponible en el momento de escribir el libro).

Si se necesita aprender versiones anteriores del lenguaje no cubiertas en esta edición, la primera edición del libro puede descargarse gratuitamente de *http://www.BruceEckel.com*, y también está en el CD adjunto a este libro.

Algo de lo que uno se dará cuenta es de que, cuando menciono versiones anteriores del lenguaje, no uso los números de sub-revisión. En este libro me referiré sólo a Java 1.0, 1.1 y 2, para protegerme de errores tipográficos producidos por sub-revisiones posteriores de estos productos.

Seminarios y mi papel como mentor

Mi empresa proporciona seminarios de formación de cinco días, en máquina, públicos e *in situ*, basados en el material de este libro. Determinado material de cada capítulo representa una lección, seguida de un periodo de ejercicios guiados de forma que cada alumno recibe atención personal. Las conferencias y las diapositivas del seminario introductorio también están en el CD ROM para proporcional al menos alguna de la experiencia del seminario sin el viaje y el coste que conllevaría. Para más información, visitar *http://www.BruceEckel.com*.

Mi compañía también proporciona consultoría, servicios de orientación y acompañamiento para ayudar a guiar un proyecto a lo largo de su ciclo de desarrollo —especialmente indicado para el primer proyecto en Java de una empresa.

Errores

Sin que importe cuántos trucos utiliza un escritor para detectar errores, siempre hay alguno que se queda ahí y que algún lector encontrará.

Hay un formulario para remitir errores al principio de cada capítulo en la versión HTML del libro (y en el CD ROM unido al final de este libro, además de descargable de *http://www.BruceEckel.com*) y también en el propio sitio web, en la página correspondiente a este libro. Si se descubre algo que uno piense que puede ser un error, por favor, utilice el formulario para remitir el error junto con la corrección sugerida. Si es necesario, incluya el archivo de código fuente original y cualquier modificación que se sugiera. Su ayuda será apreciada.

Nota sobre el diseño de la portada

La portada de *Piensa en Java* está inspirada en el *American Arts & Crafts Movement*, que se fundó al cambiar de siglo y alcanzó su cenit entre los años 1900 y 1920. Empezó en Inglaterra como una reacción tanto a la producción de las máquinas de la Revolución Industrial y al estilo victoriano, excesivamente ornamental. *Arts & Crafts* hacía especial énfasis en el mero diseño, en las formas de la naturaleza tal y como se ven en el movimiento del *Art Nouveau*, las manualidades y la importancia del trabajo individual, y sin embargo sin renunciar al uso de herramientas modernas. Hay muchas réplicas con la situación de hoy en día: el cambio de siglo, la evolución de los principios puros de la revolución de los computadores a algo más refinado y más significativo para las personas individuales, y el énfasis en el arte individual que hay en el software, frente a su simple manufactura.

Veo Java de esta misma forma: como un intento de elevar al programador más allá de la mecánica de un sistema operativo y hacia el "arte del software".

Tanto el autor como el diseñador del libro/portada (que han sido amigos desde la infancia) encuentran la inspiración en este movimiento, y ambos poseen muebles, lámparas y otros elementos que o bien son originales, o bien están inspirados en este periodo.

El otro tema de la cubierta sugiere una caja de colecciones que podría usar un naturalista para mostrar los especímenes de insectos que ha guardado. Estos insectos son objetos, ubicados dentro de la caja de objetos. Los objetos caja están a su vez ubicados dentro del "objeto cubierta", que ilustra el concepto fundamental de la agregación en la programación orientada a objetos. Por supuesto, un programador no puede ayudar si no es produciendo "errores" en la asociación, y aquí los errores se han capturado siendo finalmente confinados en una pequeña caja de muestra, como tratando de mostrar la habilidad de Java para encontrar, mostrar y controlar los errores (lo cual es sin duda uno de sus más potentes atributos).

Agradecimientos

En primer lugar, gracias a los asociados que han trabajado conmigo para dar seminarios, proporcionar consultoría y desarrollar productos de aprendizaje: Andrea Provaglio, Dave Bastlett (que también contribuyó significativamente al Capítulo 15), Bill Venners y Larry O'Brien. Aprecio vuestra paciencia a medida que sigo intentando desarrollar el mejor modelo para que tipos tan independientes como nosotros podamos trabajar juntos. Gracias a Rolf André Klaedtke (Suiza); Martin Vleck, Martin Byer, Vlada & Pavel Lahoda, Martin el Oso, y Hanka (Praga); y a Marco Cantu (Italia) por darme alojamiento durante mi primera gira seminario auto organizada por Europa.

Gracias a la *Doyle Street Cohousing Community* por soportarme durante los dos años que me llevó escribir la primera edición de este libro (y por aguantarme en general). Muchas gracias a Kevin y Sonda Donovan por subarrendarme su magnífico lugar en Creste Butte, Colorado, durante el verano mientras trabajaba en la primera edición del libro. Gracias también a los amigables residentes de Crested Butte y al *Rocky Mountain Biologial Laboratory* que me hizo sentir tan acogido.

Gracias a Claudette Moore de la *Moore Literary Agency* por su tremenda paciencia y perseverancia a la hora de lograr que yo hiciera exactamente lo que yo quería hacer.

Mis dos primeros libros se publicaron con Jeff Pepper como editor de Osborne/McGraw-Hill. Jeff apareció en el lugar oportuno y en la hora oportuna en Prentice-Hall y me ha allanado el camino y ha hecho que ocurra todo lo que tenía que ocurrir para que ésta se convirtiera en una experiencia de publicación agradable. Gracias, Jeff —significa mucho para mí.

Estoy especialmente en deuda con Gen Kiyooka y su compañía, Digigami, que me proporcionó gentilmente mi primer servidor web durante los muchos años iniciales de presencia en la Web. Esto constituyó una ayuda de valor incalculable.

Gracias a Cay Hostmann (coautor de *Core Java*, Prentice-Hall, 2000), D'Arcy Smith (Symantec) y Paul Tyma (coautor de *Java Primer Plus*, The Waite Group, 1996), por ayudarme a aclarar conceptos sobre el lenguaje.

Gracias a la gente que ha hablado en mi curso de Java en la *Software Development Conference*, y a los alumnos de mis cursos, que realizan las preguntas que necesito oír para poder hacer un material más claro.

Gracias espaciales a Larry y Tina O'Brien, que me ayudaron a volcar mis seminarios en el CD ROM original *Hands-On Java*. (Puede encontrarse más información en *http://www.BruceEckel.com.*)

Mucha gente me envió correcciones y estoy en deuda con todos ellos, pero envío gracias en particular a (por la primera edición): Kevin Raulerson (encontró cientos de errores enormes), Bob Resendes (simplemente increíble), John Pinto, Joe Dante, Jose Sharp (los tres son fabulosos), David Coms (muchas correcciones gramaticales y aclaraciones), Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennos P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson, y otros muchos. El profesor Marc Meurrens puso gran cantidad de esfuerzo en publicitar y hacer disponible la versión electrónica de la primera edición del libro en toda Europa.

Ha habido muchísimos técnicos en mi vida que se han convertido en amigos y que también han sido, tanto influyentes, como inusuales por el hecho de que hacen yoga y practican otras formas de ejercicio espiritual, que yo también encuentro muy instructivo e inspirador. Son Karig Borckschmidt, Gen Kiyooka y Andrea Provaglio, (que ayuda en el entendimiento de Java y en la programación general en Italia, y ahora en los Estados Unidos como un asociado del equipo MindView).

No es que me haya sorprendido mucho que entender Delphi me ayudara a entender Java, pues tienen muchas decisiones de diseño del lenguaje en común. Mis amigos de Delphi me proporcionaron ayuda facilitándome a alcanzar profundidad en este entorno de programación tan maravilloso. Son Marco Cantu (otro italiano —¿quizás aprender Latín es una ayuda para entender los lenguajes de programación?), Neil Rubenking (que solía hacer yoga, era vegetariano,... hasta que descubrió los computadores) y por supuesto, Zack Urlocker, un colega de hace tiempo con el que me he movido por todo el mundo.

Las opiniones y el soporte de mi amigo Richard Hale Shaw han sido de mucha ayuda (y la de Kim también). Richard y yo pasamos muchos meses dando seminarios juntos e intentando averiguar cuál era la experiencia de aprendizaje perfecta desde el punto de vista de los asistentes. Gracias a KoAnn Vikoren, Eric Faurot, Marco Pardi, y el resto de equipo y tripulación de MFI. Gracias especialmente a Tara Arrowood, que me volvió a inspirar en las posibilidades de las conferencias.

El diseño del libro, de la portada, y la foto de ésta fueron creadas por mi amigo Daniel Hill-Harris, autor y diseñador de renombre (*http://www.Wil-Harris.com*), que solía jugar con letras de goma en el colegio mientras esperaba a que se inventaran los computadores y los ordenadores personales, y se quejaba de que yo siempre estuviera enmarañado con mis problemas de álgebra. Sin embargo, he producido páginas listas para la cámara por mí mismo, por lo que los errores de tipografía son míos. Para escribir el libro se usó Microsoft ® Word 97 for Windows, y para crear páginas listas para fotografiar en Adobe Acrobat; el libro se creó directamente a partir de los ficheros Acrobat PDF. (Como un tributo a la edad electrónica, estuve fuera en las dos ocasiones en que se produjo la versión final del libro —la primera edición se envío desde Capetown, Sudáfrica, y la segunda edición se

envío desde Praga.) La tipología del cuerpo es *Georgia* y los títulos están en *Verdana*. La tipografía de la portada es *ITC Rennie Mackintosch*.

Gracias a los vendedores que crearon los compiladores. Borland, el Blackdown Group (para Linux), y por supuesto, Sun.

Gracias especiales a todos mis profesores y alumnos (que son a su vez mis profesores). La persona que me enseñó a escribir fue Gabrielle Rico (autora de *Writing the Natural Way*, Putnam, 1985). Siempre guardaré como un tesoro aquella terrorífica semana en Esalen.

El conjunto de amigos que me han ayudado incluyen, sin ser los únicos a: Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinckely Barr, Hill Gates de *Midnight Engineering Magazine*, Larry Constantine y Lucy Lockwood, Grez Perry, Dan Putterman, Christi Westphal, GeneWang, Dave Mayer, David Intersiomne, Andrea Rosenfield, Claire Sawyers, más italianos (Laura Fallai, Corrado, ILSA, y Cristina Guistozzi). Chris y Laura Strand, los Almquists, Brad Jerbic, Marilyn Cvitanic, los Mabrys, los Haflingers, los Pollocks, Peter Vinci, las familias Robbins, las familias Moelter (y los McMillans), Michael Wilk, Dave Stoner, Laurie Adams, los Cranstons, Larry Fogg, Mike y Karen Sequeiro, Gary Entsminger y Allison Brody, Kevin Donovan y Sonda Eastlack, Chester y Shannon Andersen, Joe Lordy, Dave y Brenda Bartlett, David Lee, los Rentschlers, los Sudeks, Dick, Patty y Lee Eckel, Lynn y Todd y sus familias. Y por supuesto, papá y mamá.

Colaboradores Internet

Gracias a aquellos que me han ayudado a reescribir los ejemplos para usar la biblioteca Swing, y por cualquier otra ayuda: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajes Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Mitin Shivaram, Malcolm Davis y todo el mundo que mostró su apoyo. Verdaderamente, esto me ayudó a dar el primer salto.

1: Introducción a los objetos

La génesis de la revolución de los computadores se encontraba en una máquina, y por ello, la génesis de nuestros lenguajes de programación tiende a parecerse a esa máquina.

Pero los computadores, más que máquinas, pueden considerarse como herramientas que permiten ampliar la mente ("bicicletas para la mente", como se enorgullece de decir Steve Jobs), además de un medio de expresión inherentemente diferente. Como resultado, las herramientas empiezan a parecerse menos a máquinas y más a partes de nuestra mente, al igual que ocurre con otros medios de expresión como la escritura, la pintura, la escultura, la animación o la filmación de películas. La programación orientada a objetos (POO) es una parte de este movimiento dirigido a utilizar los computadores como si de un medio de expresión se tratara.

Este capítulo introducirá al lector en los conceptos básicos de la POO, incluyendo un repaso a los métodos de desarrollo. Este capítulo y todo el libro, toman como premisa que el lector ha tenido experiencia en algún lenguaje de programación procedural (por procedimientos), sea C u otro lenguaje. Si el lector considera que necesita una preparación mayor en programación y/o en la sintaxis de C antes de enfrentarse al presente libro, se recomienda hacer uso del CD ROM formativo *Thinking in C: Foundations for C++ and Java*, que se adjunta con el presente libro, y que puede encontrarse también la URL, *http://www.BruceEckel.com*.

Este capítulo contiene material suplementario, o de trasfondo (*background*). Mucha gente no se siente cómoda cuando se enfrenta a la programación orientada a objetos si no entiende su contexto, a grandes rasgos, previamente. Por ello, se presentan aquí numerosos conceptos con la intención de proporcionar un repaso sólido a la POO. No obstante, también es frecuente encontrar a gente que no acaba de comprender los conceptos hasta que tiene acceso a los mecanismos; estas personas suelen perderse si no se les ofrece algo de código que puedan manipular. Si el lector se siente identificado con este último grupo, estará ansioso por tomar contacto con el lenguaje en sí, por lo que debe sentirse libre de saltarse este capítulo —lo cual no tiene por qué influir en la comprensión que finalmente se adquiera del lenguaje o en la capacidad de escribir programas en él mismo. Sin embargo, tarde o temprano tendrá necesidades ocasionales de volver aquí, para completar sus nociones en aras de lograr una mejor comprensión de la importancia de los objetos y de la necesidad de comprender cómo acometer diseños haciendo uso de ellos.

El progreso de la abstracción

Todos los lenguajes de programación proporcionan abstracciones. Puede incluso afirmarse que la complejidad de los problemas a resolver es directamente proporcional a la clase (tipo) y calidad de las abstracciones a utilizar, entendiendo por tipo "clase", *aquello que se desea abstraer*. El lenguaje

ensamblador es una pequeña abstracción de la máquina subyacente. Muchos de los lenguajes denominados "imperativos" desarrollados a continuación del antes mencionado ensamblador (como Fortran, BASIC y C) eran abstracciones a su vez del lenguaje citado. Estos lenguajes supusieron una gran mejora sobre el lenguaje ensamblador, pero su abstracción principal aún exigía pensar en términos de la estructura del computador más que en la del problema en sí a resolver. El programador que haga uso de estos lenguajes debe establecer una asociación entre el modelo de la máquina (dentro del "espacio de la solución", que es donde se modela el problema, por ejemplo, un computador) y el modelo del problema que de hecho trata de resolver (en el "espacio del problema", que es donde de hecho el problema existe). El esfuerzo necesario para establecer esta correspondencia, y el hecho de que éste no es intrínseco al lenguaje de programación, es causa directa de que sea difícil escribir programas, y de que éstos sean caros de mantener, además de fomentar, como efecto colateral (lateral), toda una la industria de "métodos de programación".

La alternativa al modelado de la máquina es modelar el problema que se trata de resolver. Lenguajes primitivos como LISP o APL eligen vistas parciales o particulares del mundo (considerando respectivamente que los problemas siempre se reducen a "listas" o a "algoritmos"). PROLOG convierte todos los problemas en cadenas de decisiones. Los lenguajes se han creado para programación basada en limitaciones o para programar exclusivamente mediante la manipulación de símbolos gráficos (aunque este último caso resultó ser excesivamente restrictivo). Cada uno de estos enfoques constituye una solución buena para determinadas clases (tipos) de problemas (aquéllos para cuya solución fueron diseñados), pero cuando uno trata de sacarlos de su dominio resultan casi impracticables.

El enfoque orientado a objetos trata de ir más allá, proporcionando herramientas que permitan al programador representar los elementos en el espacio del problema. Esta representación suele ser lo suficientemente general como para evitar al programador limitarse a ningún tipo de problema específico. Nos referiremos a elementos del espacio del problema, denominando "objetos" a sus representaciones dentro del espacio de la solución (por supuesto, también serán necesarios otros objetos que no tienen su análogo dentro del espacio del problema). La idea es que el programa pueda autoadaptarse al lingo del problema simplemente añadiendo nuevos tipos de objetos, de manera que la mera lectura del código que describa la solución constituya la lectura de palabras que expresan el problema. Se trata, en definitiva, de una abstracción del lenguaje mucho más flexible y potente que cualquiera que haya existido previamente. Por consiguiente, la POO permite al lector describir el problema en términos del propio problema, en vez de en términos del sistema en el que se ejecutará el programa final. Sin embargo, sigue existiendo una conexión con el computador, pues cada objeto puede parecer en sí un pequeño computador; tiene un estado, y se le puede pedir que lleve a cabo determinadas operaciones. No obstante, esto no quiere decir que nos encontremos ante una mala analogía del mundo real, al contrario, los objetos del mundo real también tienen características y comportamientos.

Algunos diseñadores de lenguajes han dado por sentado que la programación orientada a objetos, de por sí, no es adecuada para resolver de manera sencilla todos los problemas de programación, y hacen referencia al uso de lenguajes de programación *multiparadigma*[1].

---

[1] N. del autor: Ver *Multiparadigm Programming in Leda*, por Timothy Budd (Addison-Wesley, 1995).

Alan Kay resumió las cinco características básicas de Smalltalk, el primer lenguaje de programación orientado a objetos que tuvo éxito, además de uno de los lenguajes en los que se basa Java. Estas características constituyen un enfoque puro a la programación orientada a objetos:

1. **Todo es un objeto**. Piense en cualquier objeto como una variable: almacena datos, permite que se le "hagan peticiones", pidiéndole que desempeñe por sí mismo determinadas operaciones, etc. En teoría, puede acogerse cualquier componente conceptual del problema a resolver (bien sean perros, edificios, servicios, etc.) y representarlos como objetos dentro de un programa.

2. **Un programa es un cúmulo de objetos que se dicen entre sí lo que tienen que hacer mediante el envío de mensajes**. Para hacer una petición a un objeto, basta con "enviarle un mensaje". Más concretamente, puede considerarse que un mensaje en sí es una petición para solicitar una llamada a una función que pertenece a un objeto en particular.

3. **Cada objeto tiene su propia memoria, constituida por otros objetos**. Dicho de otra manera, uno crea una nueva clase de objeto construyendo un paquete que contiene objetos ya existentes. Por consiguiente, uno puede incrementar la complejidad de un programa, ocultándola tras la simplicidad de los propios objetos.

4. **Todo objeto es de algún tipo**. Cada objeto es *un elemento de una clase*, entendiendo por "clase" un sinónimo de "tipo". La característica más relevante de una clase la constituyen "el conjunto de mensajes que se le pueden enviar".

5. **Todos los objetos de determinado tipo pueden recibir los mismos mensajes**. Ésta es una afirmación de enorme trascendencia como se verá más tarde. Dado que un objeto de tipo "círculo" es también un objeto de tipo "polígono", se garantiza que todos los objetos "círculo" acepten mensajes propios de "polígono". Esto permite la escritura de código que haga referencia a polígonos, y que de manera automática pueda manejar cualquier elemento que encaje con la descripción de "polígono". Esta capacidad de *suplantación* es uno de los conceptos más potentes de la POO.

Todo objeto tiene una interfaz

Aristóteles fue probablemente el primero en estudiar cuidadosamente el concepto de *tipo*; hablaba de "la clase de los pescados o la clase de los peces". La idea de que todos los objetos, aún siendo únicos, son también parte de una clase de objetos, todos ellos con características y comportamientos en común, ya fue usada en el primer lenguaje orientado a objetos, Simula-67, que ya incluye la palabra clave **clase**, que permite la introducción de un nuevo tipo dentro de un programa.

Simula, como su propio nombre indica, se creó para el desarrollo de simulaciones, como el clásico del cajero de un banco, en el que hay cajero, clientes, cuentas, transacciones y unidades monetarias —un montón de "objetos". Todos los objetos que, con excepción de su estado, son idénticos durante la ejecución de un programa se agrupan en "clases de objetos", que es precisamente de donde proviene la palabra clave **clase**. La creación de tipos abstractos de datos (clases) es un concepto fun-

damental en la programación orientada a objetos. Los tipos abstractos de datos funcionan casi como los tipos de datos propios del lenguaje: es posible la creación de variables de un tipo (que se denominan *objetos* o *instancias* en el dialecto propio de la orientación a objetos) y manipular estas variables (mediante el *envío* o *recepción de mensajes*; se envía un mensaje a un objeto y éste averigua qué debe hacer con él). Los miembros (elementos) de cada clase comparten algunos rasgos comunes: toda cuenta tiene un saldo, todo cajero puede aceptar un ingreso, etc. Al mismo tiempo, cada miembro tiene su propio estado, cada cuenta tiene un saldo distinto, cada cajero tiene un nombre. Por consiguiente, los cajeros, clientes, cuentas, transacciones, etc. también pueden ser representados mediante una entidad única en el programa del computador. Esta entidad es el objeto, y cada objeto pertenece a una clase particular que define sus características y comportamientos.

Por tanto, aunque en la programación orientada a objetos se crean nuevos tipos de datos, virtualmente todos los lenguajes de programación orientada a objetos hacen uso de la palabra clave "clase". Siempre que aparezca la palabra clave "tipo" (*type*) puede sustituirse por "clase" (*class*) y viceversa[2].

Dado que una clase describe a un conjunto de objetos con características (datos) y comportamientos (funcionalidad) idénticos, una clase es realmente un tipo de datos, porque un número en coma flotante, por ejemplo, también tiene un conjunto de características y comportamientos. La diferencia radica en que un programador define una clase para que encaje dentro de un problema en vez de verse forzado a utilizar un tipo de datos existente que fue diseñado para representar una unidad de almacenamiento en una máquina. Es posible extender el lenguaje de programación añadiendo nuevos tipos de datos específicos de las necesidades de cada problema. El sistema de programación acepta las nuevas clases y las cuida, y asigna las comprobaciones que da a los tipos de datos predefinidos.

El enfoque orientado a objetos no se limita a la construcción de simulaciones. Uno puede estar de acuerdo o no con la afirmación de que todo programa es una simulación del sistema a diseñar, mientras que las técnicas de POO pueden reducir de manera sencilla un gran conjunto de problemas a una solución simple.

Una vez que se establece una clase, pueden construirse tantos objetos de esa clase como se desee, y manipularlos como si fueran elementos que existen en el problema que se trata de resolver. Sin duda, uno de los retos de la programación orientada a objetos es crear una correspondencia uno a uno entre los elementos del espacio del problema y los objetos en el espacio de la solución.

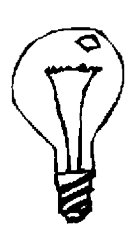
Pero, ¿cómo se consigue que un objeto haga un trabajo útil para el programador? Debe de haber una forma de hacer peticiones al objeto, de manera que éste desempeñe alguna tarea, como completar una transacción, dibujar algo en la pantalla o encender un interruptor. Además, cada objeto sólo puede satisfacer determinadas peticiones. Las peticiones que se pueden hacer a un objeto se encuentran definidas en su *interfaz*, y es el tipo de objeto el que determina la interfaz. Un ejemplo sencillo sería la representación de una bombilla:

---

[2] Algunas personas establecen una distinción entre ambos, remarcando que un tipo determina la interfaz, mientras que una clase es una implementación particular de una interfaz.

Tipo **Luz**

Interfaz encender() apagar() brillar() atenuar()



```
Luz lz = new Luz();
lz.encender();
```

La interfaz establece *qué* peticiones pueden hacerse a un objeto particular. Sin embargo, debe hacer código en algún lugar que permita satisfacer esas peticiones. Éste, junto con los datos ocultos, constituye la *implementación*. Desde el punto de vista de un lenguaje de programación procedural, esto no es tan complicado. Un tipo tiene una función asociada a cada posible petición, de manera que cuando se hace una petición particular a un objeto, se invoca a esa función. Este proceso se suele simplificar diciendo que se ha "enviado un mensaje" (hecho una petición) a un objeto, y el objeto averigua qué debe hacer con el mensaje (ejecuta el código).

Aquí, el nombre del tipo o clase es **Luz**, el nombre del objeto **Luz** particular es **lz**, y las peticiones que pueden hacerse a una **Luz** son encender, apagar, brillar o atenuar. Es posible crear una **Luz** definiendo una "referencia" (**lz**) a ese objeto e invocando a **new** para pedir un nuevo objeto de ese tipo. Para enviar un mensaje al objeto, se menta el nombre del objeto y se conecta al mensaje de petición mediante un punto. Desde el punto de vista de un usuario de una clase predefinida, éste es el no va más de la programación con objetos.

El diagrama anteriormente mostrado sigue el formato del Lenguaje de Modelado Unificado o *Unified Modeling Language* (UML). Cada clase se representa mediante una caja, en la que el nombre del tipo se ubica en la parte superior, cualquier dato necesario para describirlo se coloca en la parte central, y las *funciones miembro* (las funciones que pertenecen al objeto) en la parte inferior de la caja. A menudo, solamente se muestran el nombre de la clase y las funciones miembro públicas en los diagramas de diseño UML, ocultando la parte central. Si únicamente interesan los nombres de las clases, tampoco es necesario mostrar la parte inferior de la caja.

La implementación oculta

Suele ser de gran utilidad descomponer el tablero de juego en *creadores de clases* (elementos que crean nuevos tipos de datos) y *programadores clientes*[3] (consumidores de clases que hacen uso de los tipos de datos en sus aplicaciones). La meta del programador cliente es hacer uso de un gran repertorio de clases que le permitan acometer el desarrollo de aplicaciones de manera rápida. La

[3] Nota del autor: Término acuñado por mi amigo Scott Meyers.

meta del creador de clases es construir una clase que únicamente exponga lo que es necesario al programador cliente, manteniendo oculto todo lo demás. ¿Por qué? Porque aquello que esté oculto no puede ser utilizado por el programador cliente, lo que significa que el creador de la clase puede modificar la parte oculta a su voluntad, sin tener que preocuparse de cualquier impacto que esta modificación pueda implicar. La parte oculta suele representar las interioridades de un objeto que podrían ser corrompidas por un programador cliente poco cuidadoso o ignorante, de manera que mientras se mantenga oculta su implementación se reducen los errores en los programas.

En cualquier relación es importante determinar los límites relativos a todos los elementos involucrados. Al crear una biblioteca, se establece una relación con el programador cliente, que es también un programador, además de alguien que está construyendo una aplicación con las piezas que se encuentran en esta biblioteca, quizás con la intención de construir una biblioteca aún mayor.

Si todos los miembros de una clase están disponibles para todo el mundo, el programador cliente puede hacer cualquier cosa con esa clase y no hay forma de imponer reglas. Incluso aunque prefiera que el programador cliente no pueda manipular alguno de los miembros de su clase, sin control de accesos, no hay manera de evitarlo. Todo se presenta desnudo al mundo.

Por ello, la primera razón que justifica el control de accesos es mantener las manos del programador cliente apartadas de las porciones que no deba manipular —partes que son necesarias para las maquinaciones internas de los tipos de datos pero que no forman parte de la interfaz que los usuarios necesitan en aras de resolver sus problemas particulares. De hecho, éste es un servicio a los usuarios que pueden así ver de manera sencilla aquello que es sencillo para ellos, y qué es lo que pueden ignorar.

La segunda razón para un control de accesos es permitir al diseñador de bibliotecas cambiar el funcionamiento interno de la clase sin tener que preocuparse sobre cómo afectará al programador cliente. Por ejemplo, uno puede implementar una clase particular de manera sencilla para simplificar el desarrollo y posteriormente descubrir que tiene la necesidad de reescribirla para que se ejecute más rápidamente. Si tanto la interfaz como la implementación están claramente separadas y protegidas, esto puede ser acometido de manera sencilla.

Java usa tres palabras clave explícitas para establecer los límites en una clase: **public**, **private** y **protected**. Su uso y significado son bastante evidentes. Estos *modificadores de acceso* determinan quién puede usar las definiciones a las que preceden. La palabra **public** significa que las definiciones siguientes están disponibles para todo el mundo. El término **private**, por otro lado, significa que nadie excepto el creador del tipo puede acceder a esas definiciones. Así, **private** es un muro de ladrillos entre el creador y el programador cliente. Si alguien trata de acceder a un miembro **private**, obtendrá un error en tiempo de compilación. La palabra clave **protected** actúa como **private**, con la excepción de que una clase heredada tiene acceso a miembros **protected** pero no a los **private**. La herencia será definida algo más adelante.

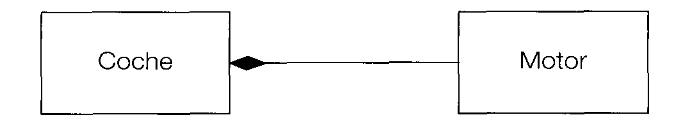
Java también tiene un "acceso por defecto", que se utiliza cuando no se especifica ninguna de las palabras clave descritas en el párrafo anterior. Este modo de acceso se suele denominar "amistoso" o *friendly* porque implica que las clases pueden acceder a los miembros amigos de otras clases que

estén en el mismo *package* o paquete, sin embargo, fuera del paquete, estos miembros amigos se convierten en **private**.

Reutilizar la implementación

Una vez que se ha creado y probado una clase, debería (idealmente) representar una unidad útil de código. Resulta que esta reutilización no es siempre tan fácil de lograr como a muchos les gustaría; producir un buen diseño suele exigir experiencia y una visión profunda de la problemática. Pero si se logra un diseño bueno, parece suplicar ser reutilizado. La reutilización de código es una de las mayores ventajas que proporcionan los lenguajes de programación orientados a objetos.

La manera más simple de reutilizar una clase es simplemente usar un objeto de esa clase directamente, pero también es posible ubicar un objeto de esa clase dentro de otra clase. Esto es lo que se denomina la "creación de un objeto miembro ". La nueva clase puede construirse a partir de un número indefinido de otros objetos, de igual o distinto tipo, en cualquier combinación necesaria para lograr la funcionalidad deseada dentro de la nueva clase. Dado que uno está construyendo una nueva clase a partir de otras ya existentes, a este concepto se le denomina *composición* (o, de forma más general, *agregación*). La composición se suele representar mediante una relación "es-parte-de", como en "el motor es una parte de un coche" ("carro" en Latinoamérica).



(Este diagrama UML indica la composición, y el rombo relleno indica que hay un coche. Normalmente, lo representaré simplemente mediante una línea, sin el diamante, para indicar una asociación[4].)

La composición conlleva una gran carga de flexibilidad. Los objetos miembros de la nueva clase suelen ser privados, haciéndolos inaccesibles a los programadores cliente que hagan uso de la clase. Esto permite cambiar los miembros sin que ello afecte al código cliente ya existente. También es posible cambiar los objetos miembros en tiempo de ejecución, para así cambiar de manera dinámica el comportamiento de un programa. La herencia, que se describe a continuación, no tiene esta flexibilidad, pues el compilador debe emplazar restricciones de tiempo de compilación en las clases que se creen mediante la herencia.

Dado que la herencia es tan importante en la programación orientada a objetos, casi siempre se enfatiza mucho su uso, de manera que un programador novato puede llegar a pensar que hay que ha-

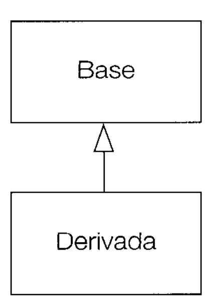
[4] Éste ya suele ser un nivel de detalle suficiente para la gran mayoría de diagramas, de manera que no es necesario indicar de manera explícita si se está utilizando agregación o composición.

cer uso de la misma en todas partes. Este pensamiento puede provocar que se elaboren diseños poco elegantes y desmesuradamente complicados. Por el contrario, primero sería recomendable intentar hacer uso de la composición, mucho más simple y sencilla. Siguiendo esta filosofía se lograrán diseños mucho más limpios. Una vez que se tiene cierto nivel de experiencia, la detección de los casos que precisan de la herencia se convierte en obvia.

Herencia: reutilizar la interfaz

Por sí misma, la idea de objeto es una herramienta más que buena. Permite empaquetar datos y funcionalidad juntos por *concepto*, de manera que es posible representar cualquier idea del espacio del problema en vez de verse forzado a utilizar idiomas propios de la máquina subyacente. Estos conceptos se expresan como las unidades fundamentales del lenguaje de programación haciendo uso de la palabra clave **class**.

Parece una pena, sin embargo, acometer todo el problema para crear una clase y posteriormente verse forzado a crear una nueva que podría tener una funcionalidad similar. Sería mejor si pudiéramos hacer uso de una clase ya existente, clonarla, y después hacer al "clon" las adiciones y modificaciones que sean necesarias. Efectivamente, esto se logra mediante la *herencia*, con la excepción de que si se cambia la clase original (denominada la *clase base*, *clase super* o *clase padre*), el "clon" modificado (denominado *clase derivada*, *clase heredada*, *subclase* o *clase hijo*) también reflejaría esos cambios.

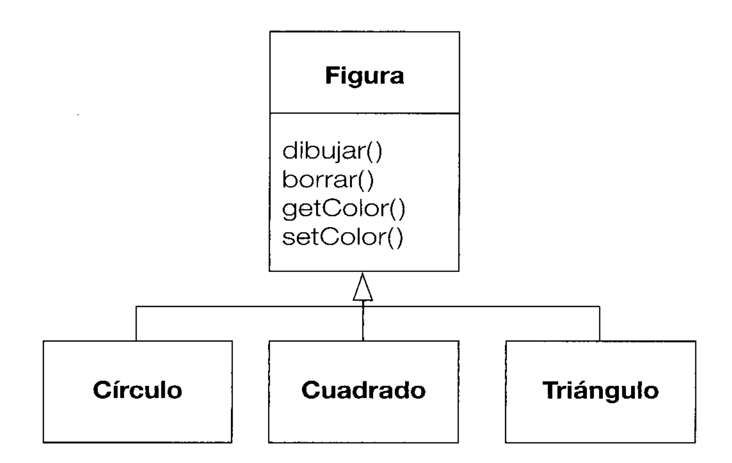


(La flecha de este diagrama UML apunta de la clase derivada a la clase base. Como se verá, puede haber más de una clase derivada.)

Un tipo hace más que definir los límites de un conjunto de objetos; también tiene relaciones con otros tipos. Dos tipos pueden tener características y comportamientos en común, pero un tipo puede contener más características que otro y también puede manipular más mensajes (o gestionarlos de manera distinta). La herencia expresa esta semejanza entre tipos haciendo uso del concepto de tipos base y tipos derivados. Un tipo base contiene todas las características y comportamientos que comparten los tipos que de él se derivan. A partir del tipo base, es posible derivar otros tipos para expresar las distintas maneras de llevar a cabo esta idea.

Por ejemplo, una máquina de reciclaje de basura clasifica los desperdicios. El tipo base es “basura”, y cada desperdicio tiene su propio peso, valor, etc. y puede ser fragmentado, derretido o descompuesto. Así, se derivan tipos de basura más específicos que pueden tener características adicionales (una botella tiene un color), o comportamientos (el aluminio se puede modelar, una lata de acero tiene capacidades magnéticas). Además, algunos comportamientos pueden ser distintos (el valor del papel depende de su tipo y condición). El uso de la herencia permite construir una jerarquía de tipos que expresa el problema que se trata de resolver en términos de los propios tipos.

Un segundo ejemplo es el clásico de la “figura geométrica” utilizada generalmente en sistemas de diseño por computador o en simulaciones de juegos. El tipo base es “figura” y cada una de ellas tiene un tamaño, color, posición, etc. Cada figura puede dibujarse, borrarse, moverse, colorearse, etc. A partir de ésta, se pueden derivar (heredar) figuras específicas: círculos, cuadrados, triángulos, etc., pudiendo tener cada uno de los cuales características y comportamientos adicionales. Algunos comportamientos pueden ser distintos, como pudiera ser el cálculo del área de los distintos tipos de figuras. La jerarquía de tipos engloba tanto las similitudes como las diferencias entre las figuras.



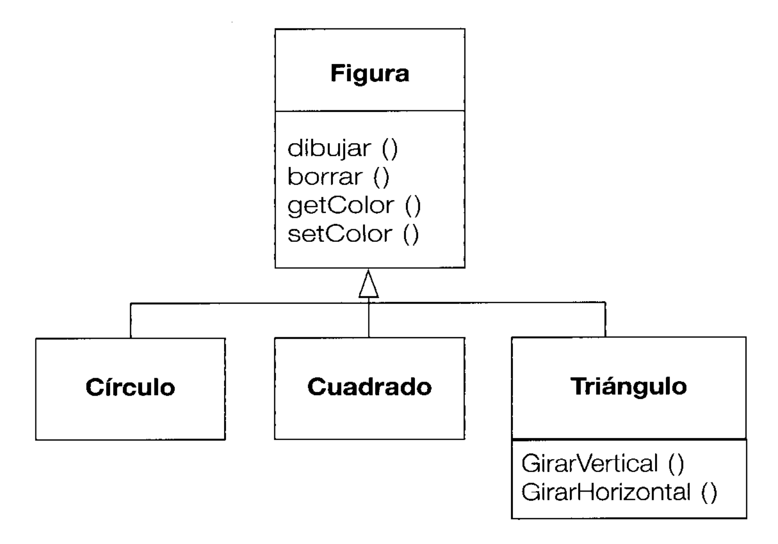
Representar la solución en los mismos términos que el problema es tremendamente beneficioso puesto que no es necesario hacer uso de innumerables modelos intermedios para transformar una descripción del problema en una descripción de la solución. Con los objetos, el modelo principal lo constituye la jerarquía de tipos, de manera que se puede ir directamente de la descripción del sistema en el mundo real a la descripción del sistema en código. Sin duda, una de las dificultades que tiene la gente con el diseño orientado a objetos es la facilidad con que se llega desde el principio al final: las mentes entrenadas para buscar soluciones completas suelen verse aturdidas inicialmente por esta simplicidad.

Al heredar a partir de un tipo existente, se crea un nuevo tipo. Este nuevo tipo contiene no sólo los miembros del tipo existente (aunque los datos privados “**private**” están ocultos e inaccesibles) sino lo que es más importante, duplica la interfaz de la clase base. Es decir, todos los mensajes que pueden ser enviados a objetos de la clase base también pueden enviarse a los objetos de la clase derivada. Dado que sabemos el tipo de una clase en base a los mensajes que se le pueden enviar, la cla-

se derivada *es del mismo tipo que la clase base*. Así, en el ejemplo anterior, "un círculo en una figura". Esta equivalencia de tipos vía herencia es una de las pasarelas fundamentales que conducen al entendimiento del significado de la programación orientada a objetos.

Dado que, tanto la clase base como la derivada tienen la misma interfaz, debe haber alguna implementación que vaya junto con la interfaz. Es decir, debe haber algún código a ejecutar cuando un objeto recibe un mensaje en particular. Si simplemente se hereda la clase sin hacer nada más, los métodos de la interfaz de la clase base se trasladan directamente a la clase derivada. Esto significa que los objetos de la clase derivada no sólo tienen el mismo tipo sino que tienen el mismo comportamiento, aunque este hecho no es particularmente interesante.

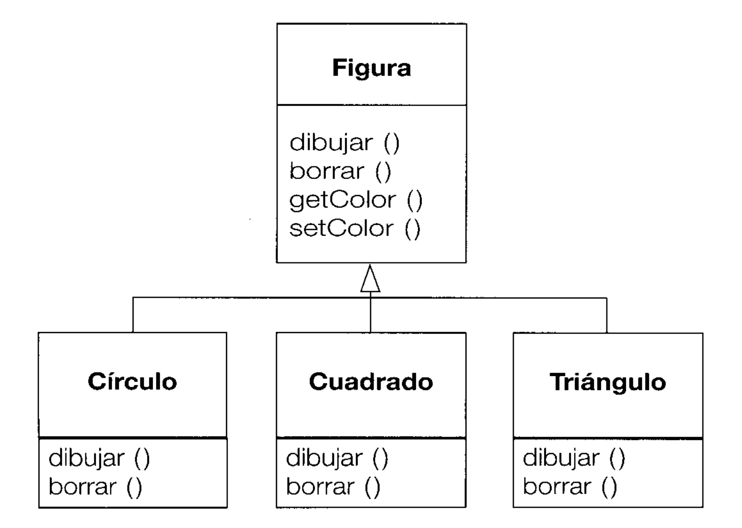
Hay dos formas de diferenciar una clase nueva derivada de la clase base original. El primero es bastante evidente: se añaden nuevas funciones a la clase derivada. Estas funciones nuevas no forman parte de la interfaz de la clase base, lo que significa que la clase base simplemente no hacía todo lo que ahora se deseaba, por lo que fue necesario añadir nuevas funciones. Ese uso simple y primitivo de la herencia es, en ocasiones, la solución perfecta a los problemas. Sin embargo, debería considerarse detenidamente la posibilidad de que la clase base llegue también a necesitar estas funciones adicionales. Este proceso iterativo y de descubrimiento de su diseño es bastante frecuente en la programación orientada a objetos.



Aunque la herencia puede implicar en ocasiones (especialmente en Java, donde la palabra clave que hace referencia a la misma es **extends**) que se van a añadir funcionalidades a una interfaz, esto no tiene por qué ser siempre así. La segunda y probablemente más importante manera de diferenciar una nueva clase es *variar* el comportamiento de una función ya existente en la clase base. A esto se le llama redefinición[5] (anulación o superposición) de la función.

Para redefinir una función simplemente se crea una nueva definición de la función dentro de la clase derivada. De esta manera puede decirse que "se está utilizando la misma función de la interfaz pero se desea que se comporte de manera distinta dentro del nuevo tipo".

[5] En el original *overriding* (N. del T.).

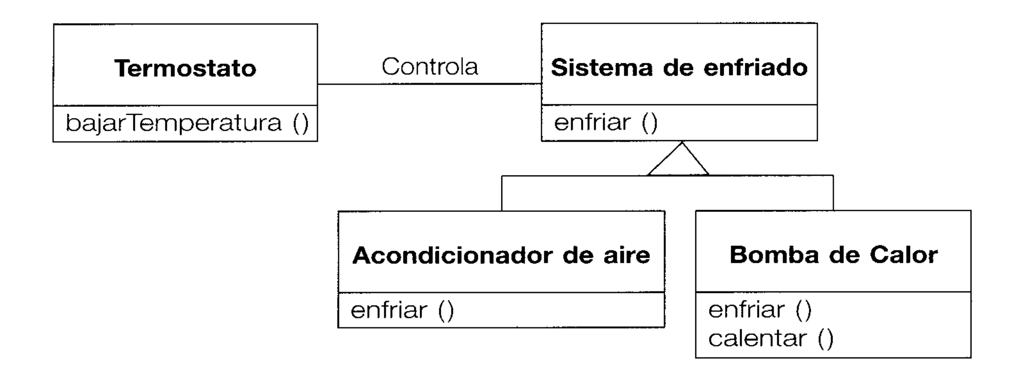


La relación es-un frente a la relación es-como-un

Es habitual que la herencia suscite un pequeño debate: ¿debería la herencia superponer *sólo* las funciones de la clase base (sin añadir nuevas funciones miembro que no se encuentren en ésta)? Esto significaría que el tipo derivado sea *exactamente* el mismo tipo que el de la clase base, puesto que tendría exactamente la misma interfaz. Consecuentemente, es posible sustituir un objeto de la clase derivada por otro de la clase base. A esto se le puede considerar *sustitución pura*, y a menudo se le llama el *principio de sustitución*. De cierta forma, ésta es la manera ideal de tratar la herencia. Habitualmente, a la relación entre la clase base y sus derivadas que sigue esta filosofía se le denomina relación *es-un*, pues es posible decir que "un círculo *es un* polígono". Una manera de probar la herencia es determinar si es posible aplicar la relación *es-un* a las clases en liza, y tiene sentido.

Hay veces en las que es necesario añadir nuevos elementos a la interfaz del tipo derivado, extendiendo así la interfaz y creando un nuevo tipo. Éste puede ser también sustituido por el tipo base, pero la sustitución no es perfecta pues las nuevas funciones no serían accesibles desde el tipo base. Esta relación puede describirse como la relación *es-como-un*[6]; el nuevo tipo tiene la interfaz del viejo pero además contiene otras funciones, así que no se puede decir que sean exactamente iguales. Considérese por ejemplo un acondicionador de aire. Supongamos que una casa está cableada y tiene las botoneras para refrescarla, es decir, tiene una interfaz que permite controlar la temperatura. Imagínese que se estropea el acondicionador de aire y se reemplaza por una bomba de calor que puede tanto enfriar como calentar. La bomba de calor *es-como-un* acondicionador de aire, pero puede hacer más funciones. Dado que el sistema de control de la casa está diseñado exclusivamente para controlar el enfriado, se encuentra restringido a la comunicación con la parte "enfriadora" del nuevo objeto. Es necesario extender la interfaz del nuevo objeto, y el sistema existente únicamente conoce la interfaz original.

[6] Término acuñado por el autor.



Por supuesto, una vez que uno ve este diseño, está claro que la clase base "sistema de enfriado" no es lo suficientemente general, y debería renombrarse a "sistema de control de temperatura" de manera que también pueda incluir calentamiento —punto en el que el principio de sustitución funcionará. Sin embargo, este diagrama es un ejemplo de lo que puede ocurrir en el diseño y en el mundo real.

Cuando se ve el principio de sustitución es fácil sentir que este principio (la sustitución pura) es la única manera de hacer las cosas, y de hecho, *es* bueno para los diseños que funcionen así. Pero hay veces que está claro que hay que añadir nuevas funciones a la interfaz de la clase derivada. Con la experiencia, ambos casos irán pareciendo obvios.

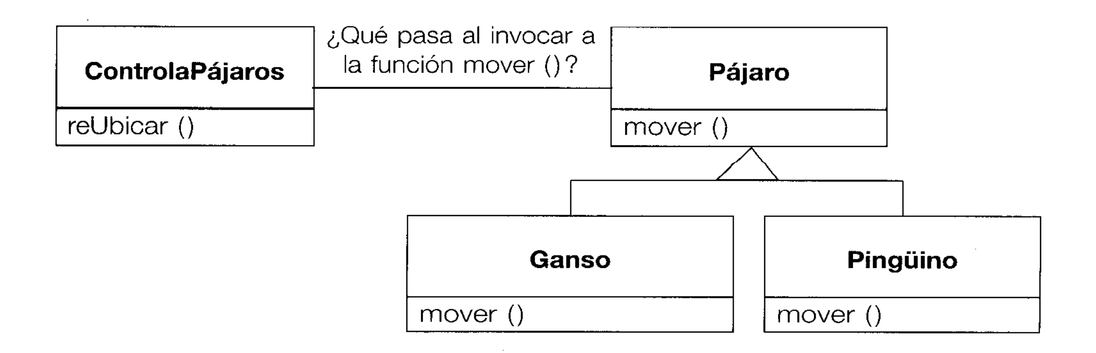
Objetos intercambiables con polimorfismo

Al tratar con las jerarquías de tipos, a menudo se desea tratar un objeto no como el tipo específico del que es, sino como su tipo base. Esto permite escribir código que no depende de tipos específicos. En el ejemplo de los polígonos, las funciones manipulan polígonos genéricos sin que importe si son círculos, cuadrados, triángulos o cualquier otro polígono que no haya sido aún definido. Todos los polígonos pueden dibujarse, borrarse y moverse, por lo que estas funciones simplemente envían un mensaje a un objeto polígono; no se preocupan de qué base hace este objeto con el mensaje.

Este tipo de código no se ve afectado por la adición de nuevos tipos, y esta adición es la manera más común de extender un programa orientado a objetos para que sea capaz de manejar nuevas situaciones. Por ejemplo, es posible derivar un nuevo subtipo de un polígono denominado pentágono sin tener que modificar las funciones que solamente manipulan polígonos genéricos. Esta capacidad para extender un programa de manera sencilla derivando nuevos subtipos es importante, ya que mejora considerablemente los diseños a la vez que reduce el coste del mantenimiento de software.

Sin embargo, hay un problema a la hora de tratar los objetos de tipos derivados como sus tipos base genéricos (los círculos como polígonos, las bicicletas como vehículos, los cormoranes como pájaros, etc.). Si una función va a decir a un polígono genérico que la dibuje, o a un vehículo genérico que se engrane, o a un pájaro genérico que se mueva, el compilador no puede determinar en tiempo de

compilación con exactitud qué fragmento de código se ejecutará. Éste es el punto clave —cuando se envía el mensaje, el programador no *quiere* saber qué fragmento de código se ejecutará; la función dibujar se puede aplicar de manera idéntica a un círculo, un cuadrado o un triángulo, y el objeto ejecutará el código correcto en función de su tipo específico. Si no es necesario saber qué fragmento de código se ejecutará, al añadir un nuevo subtipo, el código que ejecute puede ser diferente sin necesidad de modificar la llamada a la función. Por consiguiente, el compilador no puede saber exactamente qué fragmento de código se está ejecutando, ¿y qué es entonces lo que hace? Por ejemplo, en el diagrama siguiente, el objeto **ControlaPájaros** trabaja con objetos **Pájaro** genéricos, y no sabe exactamente de qué tipo son. Esto es conveniente para la perspectiva de **ControlaPájaros** pues no tiene que escribir código especial para determinar el tipo exacto de **Pájaro** con el que está trabajando, ni el comportamiento de ese **Pájaro**. Entonces, ¿cómo es que cuando se invoca a **mover( )** ignorando el tipo específico de **Pájaro** se dará el comportamiento correcto (un **Ganso** corre, vuela o nada, y un **Pingüino** corre o nada)?



La respuesta es una de las principales novedades en la programación orientada a objetos: el compilador no puede hacer una llamada a función en el sentido tradicional. La llamada a función generada por un compilador no-POO hace lo que se denomina una *ligadura temprana*, un término que puede que el lector no haya visto antes porque nunca pensó que se pudiera hacer de otra forma. Esto significa que el compilador genera una llamada a una función con nombre específico, y el montador resuelve esta llamada a la dirección absoluta del código a ejecutar. En POO, el programa no puede determinar la dirección del código hasta tiempo de ejecución, por lo que es necesario otro esquema cuando se envía un mensaje a un objeto genérico.

Para resolver el problema, los lenguajes orientados a objetos usan el concepto de *ligadura tardía*. Al enviar un mensaje a un objeto, no se determina el código invocado hasta tiempo de ejecución. El compilador se asegura de que la función exista y hace la comprobación de tipos de los argumentos y del valor de retorno (un lenguaje en el que esto no se haga así se dice que es *débilmente tipificado*), pero se desconoce el código exacto a ejecutar.

Para llevar a cabo la ligadura tardía, Java utiliza un fragmento de código especial en vez de la llamada absoluta. Este código calcula la dirección del cuerpo de la función utilizando información almacenada en el objeto (este proceso se relata con detalle en el Capítulo 7). Por consiguiente, cada objeto puede comportarse de manera distinta en función de los contenidos de ese fragmento de có-

digo especial. Cuando se envía un mensaje a un objeto, éste, de hecho, averigua qué es lo que debe hacer con ese mensaje.

En algunos lenguajes (C++ en particular) debe establecerse explícitamente que se desea que una función tenga la flexibilidad de las propiedades de la ligadura tardía. En estos lenguajes, por defecto, la correspondencia con las funciones miembro no se establece dinámicamente, por lo que es necesario recordar que hay que añadir ciertas palabras clave extra para lograr el polimorfismo.

Considérese el ejemplo de los polígonos. La familia de clases (todas ellas basadas en la misma interfaz uniforme) ya fue representada anteriormente en este capítulo. Para demostrar el polimorfismo, se escribirá un único fragmento de código que ignora los detalles específicos de tipo y solamente hace referencia a la clase base. El código está *desvinculado* de información específica del tipo, y por consiguiente es más fácil de escribir y entender. Y si se añade un nuevo tipo —por ejemplo un **Hexágono**— mediante herencia, el código que se escriba trabajará tan perfectamente con el nuevo **Polígono** como lo hacía con los tipos ya existentes, y por consiguiente, el programa es *ampliable*. Si se escribe un método en Java (y pronto aprenderá el lector a hacerlo):

```
void hacerAlgo (Poligono p) {
p.borrar();
// ...
p.dibujar();
}
```

Esta función se entiende con cualquier **Polígono**, y por tanto, es independiente del tipo específico de objeto que esté dibujando y borrando. En cualquier otro fragmento del programa se puede usar la función **hacerAlgo( )**:

```
Circulo c = new Circulo();
Triangulo t = new Triangulo ();
Linea l = new Linea();
hacerAlgo(c);
hacerAlgo(t);
hacerAlgo(l);
```

Las llamadas a **hacerAlgo( )** trabajan correctamente, independientemente del tipo de objeto.

De hecho, éste es un truco bastante divertido. Considérese la línea:

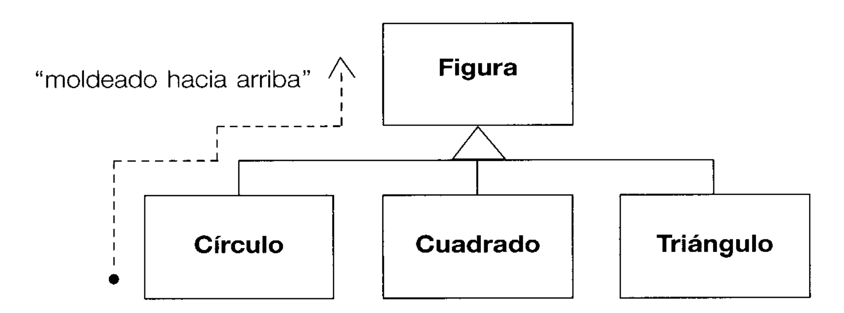
```
hacerAlgo(c);
```

Lo que está ocurriendo es que se está pasando un **Círculo** a una función que espera un **Polígono**. Como un **Círculo** *es un* **Polígono**, puede ser tratado como tal por **hacerAlgo( )**. Es decir, cualquier mensaje que **hacerAlgo( )** pueda enviar a un **Polígono**, también podrá aceptarlo un **Círculo**. Por tanto, obrar así es algo totalmente seguro y lógico.

A este proceso de tratar un tipo derivado como si fuera el tipo base se le llama conversión de tipos (*moldeado*) *hacia arriba*[7]. El nombre moldear (cast) se utiliza en el sentido de moldear (convertir) un

[7] En el original inglés, *casting*.

molde, y es *hacia arriba* siguiendo la manera en que se representa en los diagramas la herencia, con el tipo base en la parte superior y las derivadas colgando hacia abajo. Por consiguiente, hacer un moldeado (*casting*) a la clase base es moverse hacia arriba por el diagrama de herencias: *moldeado hacia arriba.*



Un programa orientado a objetos siempre tiene algún moldeado hacia arriba pues ésta es la manera de desvincularse de tener que conocer el tipo exacto con que se trabaja en cada instante. Si se echa un vistazo al código de **hacerAlgo( )**:

```
p.borrar();
// ...
p.dibujar();
```

Obsérvese que no se dice "caso de ser un **Círculo**, hacer esto; caso de ser un **Cuadrado**, hacer esto otro, etc.". Si se escribe código que compruebe todos los tipos posibles que puede ser un **Polígono**, el tipo de código se complica, además de hacerse necesario modificarlo cada vez que se añade un nuevo tipo de **Polígono**. Aquí, simplemente se dice que "en el caso de los polígonos, se sabe que es posible aplicarles las operaciones de **borrar( )** y **dibujar( )**, eso sí, teniendo en cuenta todos los detalles de manera correcta".

Lo que más llama la atención del código de **hacerAlgo( )** es que, de alguna manera, se hace lo correcto. Invocar a **dibujar( )** para **Círculo** hace algo distinto que invocar a **dibujar( )** para **Cuadrado** o **Línea**, pero cuando se envía el mensaje **dibujar( )** a un **Polígono** anónimo, se da el comportamiento correcto basándose en el tipo actual de **Polígono**. Esto es sorprendente porque, como se mencionó anteriormente, cuando el compilador de Java está compilando el código de **hacerAlgo( )**, no puede saber exactamente qué tipos está manipulando. Por ello, generalmente, se espera que acabe invocando a la versión de **borrar( )** y **dibujar( )** de la clase base **Polígono** y no a las específicas de **Círculo**, **Cuadrado** y **Línea**. Y sigue ocurriendo lo correcto por el polimorfismo. El compilador y el sistema de tiempo real se hacen cargo de los detalles; todo lo que hace falta saber es qué ocurre, y lo que es más importante, cómo diseñar haciendo uso de ello. Al enviar un mensaje a un objeto, el objeto hará lo correcto, incluso cuando se vea involucrado el moldeado hacia arriba.

Clases base abstractas e interfaces

A menudo es deseable que la clase base *únicamente* presente una interfaz para sus clases derivadas. Es decir, no se desea que nadie cree objetos de la clase base, sino que sólo se hagan moldeados ha-

cia arriba de la misma de manera que se pueda usar su interfaz. Esto se logra convirtiendo esa clase en *abstracta* usando la palabra clave **abstract**. Si alguien trata de construir un objeto de una clase **abstracta** el compilador lo evita. Esto es una herramienta para fortalecer determinados diseños.

También es posible utilizar la palabra clave **abstract** para describir un método que no ha sido aún implementado —indicando "he aquí una función interfaz para todos los tipos que se hereden de esta clase, pero hasta la fecha no existe una implementación de la misma". Se puede crear un método **abstracto** sólo dentro de una clase abstracta. Cuando se hereda la clase, debe implementarse el método o de lo contrario también la clase heredada se convierte en **abstracta**. La creación de métodos **abstractos** permite poner un método en una interfaz sin verse forzado a proporcionar un fragmento de código, posiblemente sin significado, para ese método.

La palabra clave **interface** toma el concepto de clase **abstracta** un paso más allá, evitando totalmente las definiciones de funciones. La **interfaz** es una herramienta muy útil y utilizada, ya que proporcionar la separación perfecta entre interfaz e implementación. Además, si se desea, es posible combinar muchos elementos juntos mientras que no es posible heredar de múltiples clases normales o abstractas.

Localización de objetos y longevidad

Técnicamente, la POO consiste simplemente en tipos de datos abstractos, herencia y polimorfismo, aunque también hay otros aspectos no menos importantes. El resto de esta sección trata de analizar esos aspectos.

Uno de los factores más importantes es la manera de crear y destruir objetos. ¿Dónde están los datos de un objeto y cómo se controla su longevidad (tiempo de vida)? En este punto hay varias filosofías de trabajo. En C++ el enfoque principal es el control de la eficiencia, proporcionando una alternativa al programador. Para lograr un tiempo de ejecución óptimo, es posible determinar el espacio de almacenamiento y la longevidad en tiempo de programación, ubicando los objetos en la pila (creando las variables *scoped* o *automatic*) o en el área de almacenamiento estático. De esta manera se prioriza la velocidad de la asignación y liberación de espacio de almacenamiento, cuyo control puede ser de gran valor en determinadas situaciones. Sin embargo, se sacrifica en flexibilidad puesto que es necesario conocer la cantidad exacta de objetos, además de su longevidad y su tipo, mientras se escribe el programa. Si se está tratando de resolver un problema más general como un diseño asistido por computador, la gestión de un almacén o el control de tráfico aéreo, este enfoque resulta demasiado restrictivo.

El segundo enfoque es crear objetos dinámicamente en un espacio de memoria denominado el montículo o montón (*heap*). En este enfoque, no es necesario conocer hasta tiempo de ejecución el número de objetos necesario, cuál es su longevidad o a qué tipo exacto pertenecen. Estos aspectos se determinarán justo en el preciso momento en que se ejecute el programa. Si se necesita un nuevo objeto, simplemente se construye en el *montículo* en el instante en que sea necesario. Dado que el almacenamiento se gestiona dinámicamente, en tiempo de ejecución, la cantidad de tiempo necesaria

para asignar espacio de almacenamiento en el montículo es bastante mayor que el tiempo necesario para asignar espacio a la pila. (La creación de espacio en la pila suele consistir simplemente en una instrucción al ensamblador que mueve hacia abajo el puntero de pila, y otra para moverlo de nuevo hacia arriba.) El enfoque dinámico provoca generalmente el pensamiento lógico de que los objetos tienden a ser complicados, por lo que la sobrecarga debida a la localización de espacio de almacenamiento y su liberación no deberían tener un impacto significativo en la creación del objeto. Es más, esta mayor flexibilidad es esencial para resolver en general el problema de programación.

Java utiliza exclusivamente el segundo enfoque[8]. Cada vez que se desea crear un objeto se usa la palabra clave **new** para construir una instancia dinámica de ese objeto.

Hay otro aspecto, sin embargo, a considerar: la longevidad de un objeto. Con los lenguajes que permiten la creación de objetos en la pila, el compilador determina cuánto dura cada objeto y puede destruirlo cuando no es necesario. Sin embargo, si se crea en el montículo, el compilador no tiene conocimiento alguno sobre su longevidad. En un lenguaje como C++ hay que determinar en tiempo de programación cuándo destruir el objeto, lo cual puede conducir a fallos de memoria si no se hace de manera correcta (y este problema es bastante común en los programas en C++). Java proporciona un recolector de basura que descubre automáticamente cuándo se ha dejado de utilizar un objeto, que puede, por consiguiente, ser destruido. Un recolector de basura es muy conveniente, al reducir el número de aspectos a tener en cuenta, así como la cantidad de código a escribir. Y lo que es más importante, el recolector de basura proporciona un nivel de seguridad mucho mayor contra el problema de los fallos de memoria (que ha hecho abandonar más de un proyecto en C++).

El resto de esta sección se centra en factores adicionales relativos a la longevidad de los objetos y su localización.

Colecciones e iteradores

Si se desconoce el número de objetos necesarios para resolver un problema en concreto o cuánto deben durar, también se desconocerá cómo almacenar esos objetos. ¿Cómo se puede saber el espacio a reservar para los mismos? De hecho, no se puede, pues esa información se desconocerá hasta tiempo de ejecución.

La solución a la mayoría de problemas de diseño en la orientación a objetos parece sorprendente: se crea otro tipo de objeto. El nuevo tipo de objeto que resuelve este problema particular tiene referencias a otros objetos. Por supuesto, es posible hacer lo mismo con un array, disponible en la mayoría de lenguajes. Pero hay más. Este nuevo objeto, generalmente llamado *contenedor* (llamado también *colección*, pero la biblioteca de Java usa este término con un sentido distinto así que este libro empleará la palabra "contenedor"), se expandirá a sí mismo cuando sea necesario para albergar cuanto se coloque dentro del contenedor. Simplemente se crea el objeto contenedor, y él se encarga de los detalles.

Afortunadamente, un buen lenguaje POO viene con un conjunto de contenedores como parte del propio lenguaje. En C++, es parte de la Biblioteca Estándar C++ (Standard C++ Library), que en oca-

[8] Los tipos primitivos, de los que se hablará más adelante, son un caso especial.

siones se denomina la *Standard Template Library, Biblioteca de plantillas estándar,* (STL). Object Pascal tiene contenedores en su *Visual Component Library* (VCL). Smalltalk tiene un conjunto de contenedores muy completo, y Java también tiene contenedores en su biblioteca estándar. En algunas bibliotecas, se considera que un contenedor genérico es lo suficientemente bueno para todas las necesidades, mientras que en otras (como en Java, por ejemplo) la biblioteca tiene distintos tipos de contenedores para distintas necesidades: un vector (denominado en Java **ArrayList**) para acceso consistente a todos los elementos, y una lista enlazada para inserciones consistentes en todos los elementos, por ejemplo, con lo que es posible elegir el tipo particular que satisface las necesidades de cada momento. Las bibliotecas de contenedores también suelen incluir conjuntos, colas, tablas de *hasing*, árboles, pilas, etc.

Todos los contenedores tienen alguna manera de introducir y extraer cosas; suele haber funciones para añadir elementos a un contenedor, y otras para extraer de nuevo esos elementos. Pero sacar los elementos puede ser más problemático porque una función de selección única suele ser restrictiva. ¿Qué ocurre si se desea manipular o comparar un conjunto de elementos del contenedor y no uno sólo?

La solución es un iterador, que es un objeto cuyo trabajo es seleccionar los elementos de dentro de un contenedor y presentárselos al usuario del iterador. Como clase, también proporciona cierto nivel de abstracción. Esta abstracción puede ser usada para separar los detalles del contenedor del código al que éste está accediendo. El contenedor, a través del iterador, se llega a abstraer hasta convertirse en una simple secuencia, que puede ser recorrida gracias al iterador sin tener que preocuparse de la estructura subyacente —es decir, sin preocuparse de si es un **ArrayList** (lista de arrays), un **LinkedList** (lista enlazado), un **Stack**, (pila) u otra cosa. Esto proporciona la flexibilidad de cambiar fácilmente la estructura de datos subyacente sin que el código de un programa se vea afectado. Java comenzó (en las versiones 1.0 y 1.1) con un iterador estándar denominado **Enumeration**, para todas sus clases contenedor. Java 2 ha añadido una biblioteca mucho más completa de contenedores que contiene un iterador denominado **Iterator** mucho más potente que el antiguo **Enumeration**.

Desde el punto de vista del diseño, todo lo realmente necesario es una secuencia que puede ser manipulada en aras de resolver un problema. Si una secuencia de un sólo tipo satisface todas las necesidades de un problema, entonces no es necesario hacer uso de distintos tipos. Hay dos razones por las que es necesaria una selección de contenedores. En primer lugar, los contenedores proporcionan distintos tipos de interfaces y comportamientos externos. Una pila tiene una interfaz y un comportamiento distintos del de las colas, que son a su vez distintas de los conjuntos y las listas. Cualquiera de éstos podría proporcionar una solución mucho más flexible a un problema. En segundo lugar, distintos contenedores tienen distintas eficiencias en función de las operaciones. El mejor ejemplo está en **ArrayList** y **LinkedList**. Ambos son secuencias sencillas que pueden tener interfaces y comportamientos externos idénticos. Pero determinadas operaciones pueden tener costes radicalmente distintos. Los accesos aleatorios a **ArrayList** tienen tiempos de acceso constante; se invierte el mismo tiempo independientemente del elemento seleccionado. Sin embargo, en una **LinkedList** moverse de elemento en elemento a lo largo de la lista para seleccionar uno al azar es altamente costoso, y es necesario muchísimo más tiempo para localizar un elemento cuanto más adelante se encuentre. Por otro lado, si se desea insertar un elemento en el medio de una secuencia, es mucho menos costoso hacerlo en un **LinkedList** que en un **ArrayList**. Ésta y otras operaciones tienen eficiencias distintas en función de la estructura de la secuencia subyacente. En la fase de diseño, podría comenzarse con una **LinkedList** y, al primar el rendimiento, cambiar a un

ArrayList. Dado que la abstracción se lleva a cabo a través de iteradores, es posible cambiar de uno a otro con un impacto mínimo en el código.

Finalmente, debe recordarse que un contenedor es sólo un espacio de almacenamiento en el que colocar objetos. Si este espacio resuelve todas las necesidades, no importa realmente cómo está implementado (concepto compartido con la mayoría de tipos de objetos). Si se está trabajando en un entorno de programación que tiene una sobrecarga inherente debido a otros factores, la diferencia de costes entre **ArrayList** y **LinkedList** podría no importar. Con un único tipo de secuencia podría valer. Incluso es posible imaginar la abstracción contenedora "perfecta", que pueda cambiar su implementación subyacente automáticamente en función de su uso.

La jerarquía de raíz única

Uno de los aspectos de la POO que se ha convertido especialmente prominente desde la irrupción de C++ es si todas las clases en última instancia deberían ser heredadas de una única clase base. En Java (como virtualmente en todos los lenguajes POO) la respuesta es "sí" y el nombre de esta última clase base es simplemente **Object**. Resulta que los beneficios de una jerarquía de raíz única son enormes.

Todos los objetos en una jerarquía de raíz única tienen una interfaz en común, por lo que en última instancia son del mismo tipo. La alternativa (proporcionada por C++) es el desconocimiento de que todo pertenece al mismo tipo fundamental. Desde el punto de vista de la retrocompatibilidad, esto encaja en el modelo de C mejor, y puede pensarse que es menos restrictivo, pero cuando se desea hacer programación orientada a objetos pura, es necesario proporcionar una jerarquía completa para lograr el mismo nivel de conveniencia intrínseco a otros lenguajes POO. Y en cualquier nueva biblioteca de clases que se adquiera, se utilizará alguna interfaz incompatible. Hacer funcionar esta nueva interfaz en un diseño lleva un gran esfuerzo (y posiblemente herencia múltiple). Así que ¿merece la pena la "flexibilidad" extra de C++? Si se necesita —si se dispone de una gran cantidad de código en C— es más que valiosa. Si se empieza de cero, otras alternativas, como Java, resultarán mucho más productivas.

Puede garantizarse que todos los objetos de una jerarquía de raíz única (como la proporcionada por Java) tienen cierta funcionalidad. Se sabe que es posible llevar a cabo ciertas operaciones básicas con todos los objetos del sistema. Una jerarquía de raíz única, junto con la creación de todos los objetos en el montículo, simplifica enormemente el paso de argumentos (uno de los temas más complicados de C++).

Una jerarquía de raíz única simplifica muchísimo la implementación de un recolector de basura (incluido en Java). El soporte necesario para el mismo puede instalarse en la clase base, y el recolector de basura podrá así enviar los mensajes apropiados a todos los objetos del sistema. Si no existiera este tipo de jerarquía ni la posibilidad de manipular un objeto a través de referencias, sería muy difícil implementar un recolector de basura.

Dado que está garantizado que en tiempo de ejecución la información de tipos está en todos los objetos, jamás será posible encontrar un objeto cuyo tipo no pueda ser determinado. Esto es especial-

mente importante con operaciones a nivel de sistema, tales como el manejo de excepciones, además de proporcionar una gran flexibilidad a la hora de programar.

Bibliotecas de colecciones y soporte al fácil manejo de colecciones

Dado que un contenedor es una herramienta de uso frecuente, tiene sentido tener una biblioteca de contenedores construidos para ser reutilizados, de manera que se puede elegir uno de la estantería y enchufarlo en un programa determinado. Java proporciona una biblioteca de este tipo, que satisface la gran mayoría de necesidades.

Moldeado hacia abajo frente a plantillas/genéricos

Para lograr que estos contenedores sean reutilizables, guardan un tipo universal en Java ya mencionado anteriormente: **Object** (*Objeto*). La jerarquía de raíz única implica que todo sea un **Object**, de forma que un contenedor que tiene objetos de tipo **Object** puede contener de todo, logrando así que los contenedores sean fácil de reutilizar.

Para utilizar uno de estos contenedores, basta con añadirle referencias a objetos y finalmente preguntar por ellas. Pero dado que el contenedor sólo guarda objetos de tipo **Object**, al añadir una referencia al contenedor, se hace un moldeado hacia arriba a **Object**, perdiendo por consiguiente su identidad. Al recuperarlo, se obtiene una referencia a **Object** y no una referencia al tipo que se había introducido. ¿Y cómo se convierte de nuevo en algo útil con la interfaz del objeto que se introdujo en el contenedor?

En este caso, también se hace uso del moldeado, pero en esta ocasión en vez de hacerlo hacia arriba siguiendo la jerarquía de las herencias hacia un tipo más general, se hace hacia abajo, hacia un tipo más específico. Esta forma de moldeado se denomina moldeado hacia abajo. Con el moldeado hacia arriba, como se sabe, un **Círculo**, por ejemplo, es un tipo de **Polígono**, con lo que este tipo de moldeado es seguro, pero lo que no se sabe es si un **Object** es un **Círculo** o un **Polígono**, por lo que no es muy seguro hacer moldeado hacia abajo a menos que se sepa exactamente qué es lo que se está manipulando.

Esto no es completamente peligroso, sin embargo, dado que si se hace un moldeado hacia abajo, a un tipo erróneo, se mostrará un error de tiempo de ejecución denominado *excepción*, que se describirá en breve. Sin embargo, al recuperar referencias a objetos de un contenedor, es necesario tener alguna manera de recordar exactamente lo que son para poder llevar a cabo correctamente un moldeado hacia abajo.

El moldeado hacia abajo y las comprobaciones en tiempo de ejecución requieren un tiempo extra durante la ejecución del programa, además de un esfuerzo extra por parte del programador. ¿No tendría sentido crear, de alguna manera, el contenedor de manera que conozca los tipos que guarda, eliminando la necesidad de hacer moldeado hacia abajo y por tanto, de que aparezca algún error? La solución la constituyen los tipos parametrizados, que son clases que el compilador puede adaptar automáticamente para que trabajen con tipos determinados. Por ejemplo, con un contenedor parametrizado, el compilador podría adaptar el propio contenedor para que solamente aceptara y per-

mitiera la recuperación de Polígonos.

Los tipos parametrizados son un elemento importante en C++, en parte porque este lenguaje no tiene una jerarquía de raíz única. En C++, la palabra clave que implementa los tipos parametrizados es "template". Java actualmente no tiene tipos parametrizados pues se puede lograr lo mismo —aunque de manera complicada— explotando la unicidad de raíz de su jerarquía. Sin embargo, una propuesta actualmente en curso para implementar tipos parametrizados utiliza una sintaxis muy semejante a las plantillas (*templates*) de C++.

El dilema de las labores del hogar: ¿quién limpia la casa?

Cada objeto requiere recursos simplemente para poder existir, fundamentalmente memoria. Cuando un objeto deja de ser necesario debe ser eliminado de manera que estos recursos queden disponibles para poder reutilizarse. En situaciones de programación sencillas la cuestión de cuándo eliminar un objeto no se antoja complicada: se crea el objeto, se utiliza mientras es necesario y posteriormente debe ser destruido. Sin embargo, no es difícil encontrar situaciones en las que esto se complica.

Supóngase, por ejemplo, que se está diseñando un sistema para gestionar el tráfico aéreo de un aeropuerto. (El mismo modelo podría funcionar también para gestionar paquetes en un almacén, o un sistema de alquiler de vídeos, o una residencia canina.) A primera vista, parece simple: construir un contenedor para albergar aviones, crear a continuación un nuevo avión y ubicarlo en el contenedor (para cada avión que aparezca en la zona a controlar). En el momento de eliminación, se borra (suprime) simplemente el objeto aeroplano correcto cuando un avión sale de la zona barrida.

Pero quizás, se tiene otro sistema para guardar los datos de los aviones, datos que no requieren atención inmediata, como la función de control principal. Quizás, se trata de un registro del plan de viaje de todos los pequeños aviones que abandonan el aeropuerto. Es decir, se dispone de un segundo contenedor de aviones pequeños, y siempre que se crea un objeto avión también se introduce en este segundo contenedor si se trata de un avión pequeño. Posteriormente, algún proceso en segundo plano lleva a cabo operaciones con los objetos de este segundo contenedor cada vez que el sistema está ocioso.

Ahora el problema se complica: ¿cómo se sabe cuándo destruir los objetos? Cuando el programa principal (el controlador) ha acabado de usar el objeto, puede que otra parte del sistema lo esté usando (o lo vaya a usar en un futuro). Este problema surge en numerosísimas ocasiones, y los sistemas de programación (como C++) en los que los objetos deben de borrarse explícitamente cuando acaba de usarlos, pueden volverse bastante complejos.

Con Java, el problema de vigilar que se libere la memoria se ha implementado en el recolector de basura (aunque no incluye otros aspectos de la supresión de objetos). El recolector "sabe" cuándo se ha dejado de utilizar un objeto y libera la memoria que ocupaba automáticamente. Esto (combinado con el hecho de que todos los objetos son heredados de la clase raíz única **Object** y con la existencia de una única forma de crear objetos, en el montículo) hace que el proceso de programar en Java sea mucho más sencillo que el hacerlo en C++. Hay muchas menos decisiones que tomar y menos obstáculos que sortear.

Los recolectores de basura frente a la eficiencia y flexibilidad

Si todo esto es tan buena idea, ¿por qué no se hizo lo mismo en C++? Bien, por supuesto, hay un precio que pagar por todas estas comodidades de programación, y este precio consiste en sobrecarga en tiempo de ejecución. Como se mencionó anteriormente, en C++ es posible crear objetos en la pila, y en este caso, éstos se eliminan automáticamente (pero no se dispone de la flexibilidad de crear tantos como se desee en tiempo de ejecución). La creación de objetos en la pila es la manera más eficiente de asignar espacio a los objetos y de liberarlo. La creación de objetos en el montículo puede ser mucho más costosa. Heredar siempre de una clase base y hacer que todas las llamadas a función sean polimórficas también conlleva un pequeño peaje. Pero el recolector de basura es un problema concreto pues nunca se sabe cuándo se va a poner en marcha y cuánto tiempo conllevará su ejecución. Esto significa que hay inconsistencias en los *ratios* (velocidad) de ejecución de los programas escritos en Java, por lo que éstos no pueden ser utilizados en determinadas situaciones, como por ejemplo, cuando el tiempo de ejecución de un programa es uniformemente crítico. (Se trata de los programas denominados generalmente de tiempo real, aunque no todos los problemas de programación en tiempo real son tan rígidos.)

Los diseñadores del lenguaje C++, trataron de ganarse a los programadores de C (en lo cual tuvieron bastante éxito), no quisieron añadir nuevas características al lenguaje que pudiesen influir en la velocidad o el uso de C++ en cualquier situación en la que los programadores pudiesen decantarse por C. Se logró la meta, pero a cambio de una mayor complejidad cuando se programa en C++. Java es más simple que C++, pero a cambio es menos eficiente y en ocasiones ni siquiera aplicable. Sin embargo, para un porcentaje elevado de problemas de programación, Java es la mejor elección.

Manejo de excepciones: tratar con errores

El manejo de errores ha sido, desde el principio de la existencia de los lenguajes de programación, uno de los aspectos más difíciles de abordar. Dado que es muy complicado diseñar un buen esquema de manejo de errores, muchos lenguajes simplemente ignoran este aspecto, pasando el problema a los diseñadores de bibliotecas que suelen contestar con soluciones a medias que funcionan en la mayoría de situaciones, pero que pueden ser burladas de manera sencilla; es decir, simplemente ignorándolas. Un problema importante con la mayoría de los esquemas de tratamiento de errores es que dependen de la vigilancia del programador de cara al seguimiento de una convención preestablecida no especialmente promovida por el propio lenguaje. Si el programador no está atento —cosa que ocurre muchas veces, especialmente si se tiene prisa— es posible olvidar estos esquemas con relativa facilidad.

El manejo de excepciones está íntimamente relacionado con el lenguaje de programación y a veces incluso con el sistema operativo. Una excepción es un objeto que es "lanzado", "arrojado"[9] desde el lugar en que se produce el error, y que puede ser "capturado" por el gestor de excepción apropiado

---

[9] N. del traductor: en inglés se emplea el verbo *throw*.

diseñado para manejar ese tipo de error en concreto. Es como si la gestión de excepciones constituyera un cauce de ejecución diferente, paralelo, que puede tomarse cuando algo va mal. Y dado que usa un cauce de ejecución distinto, no tiene por qué interferir con el código de ejecución normal. De esta manera el código es más simple de escribir puesto que no hay que estar comprobando los errores continuamente. Además, una excepción lanzada no es como un valor de error devuelto por una función, o un indicador (bandera) que una función pone a uno para indicar que se ha dado cierta condición de error (éstos podrían ser ignorados). Una excepción no se puede ignorar, por lo que se garantiza que será tratada antes o después. Finalmente, las excepciones proporcionan una manera de recuperarse de manera segura de una situación anormal. En vez de simplemente salir, muchas veces es posible volver a poner las cosas en su sitio y reestablecer la ejecución del programa, logrando así que éstos sean mucho más robustos.

El manejo de excepciones de Java destaca entre los lenguajes de programación pues en Java, éste se encuentra imbuido desde el principio, y es obligatorio utilizarlo. Si no se escribe un código de manera que maneje excepciones correctamente, se obtendrá un mensaje de error en tiempo de compilación. Esta garantía de consistencia hace que la gestión de errores sea mucho más sencilla.

Es importante destacar el hecho de que el manejo de excepciones no es una característica orientada a objetos, aunque en los lenguajes orientados a objetos las excepciones se suelan representar mediante un objeto. El manejo de excepciones existe desde antes de los lenguajes orientados a objetos.

Multihilo

Un concepto fundamental en la programación de computadores es la idea de manejar más de una tarea en cada instante. Muchos problemas de programación requieren que el programa sea capaz de detener lo que esté haciendo, llevar a cabo algún otro problema, y volver a continuación al proceso principal. Se han buscando múltiples soluciones a este problema. Inicialmente, los programadores que tenían un conocimiento de bajo nivel de la máquina, escribían rutinas de servicio de interrupciones, logrando la suspensión del proceso principal mediante interrupciones hardware. Aunque este enfoque funcionaba bastante bien, era dificultoso y no portable, por lo que causaba que transportar un programa a una plataforma distinta de la original fuera lento y caro.

A veces, es necesario hacer uso de las interrupciones para el manejo de tareas críticas en el tiempo, pero hay una gran cantidad de problemas en los que simplemente se intenta dividir un problema en fragmentos de código que pueden ser ejecutados por separado, de manera que se logra un menor tiempo de respuesta para todo el programa en general. Dentro de un programa, estos fragmentos de código que pueden ser ejecutados por separado, se denominan hilos, y el concepto general se denomina *multihilos*. Un ejemplo común de aplicación multihilo es la interfaz de usuario. Gracias al uso de hilos, un usuario puede presionar un botón y lograr una respuesta rápida en vez de verse forzado a esperar a que el programa acabe su tarea actual.

Normalmente, los hilos no son más que una herramienta para facilitar la planificación en un monoprocesador. Pero si el sistema operativo soporta múltiples procesadores, es posible asignar cada hilo a un procesador distinto de manera que los hilos se ejecuten verdaderamente en paralelo. Uno de los aspectos más destacables de la programación multihilo es que el programador no tiene que pre-

ocuparse de si hay uno o varios procesadores. El programa se divide de forma lógica en hilos y si hay más de un procesador, se ejecuta más rápidamente, sin que sea necesario llevar a cabo ningún ajuste adicional sobre el código.

Todo esto hace que el manejo de hilos parezca muy sencillo. Hay un inconveniente: los recursos compartidos. Si se tiene más de un hilo en ejecución tratando de acceder al mismo recurso, se plantea un problema. Por ejemplo, dos procesos no pueden enviar simultáneamente información a una impresora. Para resolver el problema, los recursos que pueden ser compartidos como la impresora, deben bloquearse mientras se están usando. Por tanto, un hilo bloquea un recurso, completa su tarea y después libera el bloqueo de manera que alguien más pueda usar ese recurso.

El hilo de Java está incluido en el propio lenguaje, lo que hace que un tema de por sí complicado se presente de forma muy sencilla. El manejo de hilos se soporta a nivel de objeto, de manera que un hilo de ejecución se representa por un objeto. Java también proporciona bloqueo de recursos limitados; puede bloquear la memoria de cualquier objeto (que en el fondo, no deja de ser un tipo de recurso compartido) de manera que sólo un objeto pueda usarlo en un instante dado. Esto se logra mediante la palabra clave **synchronized**. Otros tipos de recursos deben ser explícitamente bloqueados por el programador, generalmente, creando un objeto que represente el bloqueo que todos los hilos deben comprobar antes de acceder al recurso.

Persistencia

Al crear un objeto, existe tanto tiempo como sea necesario, pero bajo ninguna circunstancia sigue existiendo una vez que el programa acaba. Si bien esta circunstancia parece tener sentido a primera vista, hay situaciones en las que sería increíblemente útil el que un objeto pudiera existir y mantener su información incluso cuando el programa ya no esté en ejecución. De esta forma, la siguiente vez que se lance el programa, el objeto estará ahí y seguirá teniendo la misma información que tenía la última vez que se ejecutó el programa. Por supuesto, es posible lograr un efecto similar escribiendo la información en un archivo o en una base de datos, pero con la idea de hacer que todo sea un objeto, sería deseable poder declarar un objeto como persistente y hacer que alguien o algo se encargue de todos los detalles, sin tener que hacerlo uno mismo.

Java proporciona soporte para "persistencia ligera", lo que significa que es posible almacenar objetos de manera sencilla en un disco para más tarde recuperarlos. La razón de que sea "ligera" es que es necesario hacer llamadas explícitas a este almacenamiento y recuperación. Además, los JavaSpaces (descritos en el Capítulo 15) proporcionan cierto tipo de almacenamiento persistente de los objetos. En alguna versión futura, podría aparecer un soporte completo para la persistencia.

Java e Internet

Si Java es, de hecho, otro lenguaje de programación de computadores entonces uno podría preguntarse por qué es tan importante y por qué debería promocionarse como un paso revolucionario en la programación de computadores. La respuesta no es obvia ni inmediata si se procede de la perspectiva de programación tradicional. Aunque Java es muy útil de cara a la solución de problemas de

programación tradicionales autónomos, también es importante por resolver problemas de programación en la World Wide Web.

¿Qué es la Web?

La Web puede parecer un gran misterio a primera vista, especialmente cuando se oye hablar de "navegar", "presencia" y "páginas iniciales" (*home pages*). Ha habido incluso una reacción creciente contra la "Internet-manía", cuestionando el valor económico y el beneficio de un movimiento tan radical. Es útil dar un paso atrás y ver lo que es realmente, pero para hacer esto es necesario entender los sistemas cliente/servidor, otro elemento de la computación lleno de aspectos que causan también confusión.

Computación cliente/servidor

La idea principal de un sistema cliente/servidor es que se dispone de un depósito (*repositorio*) central de información —cierto tipo de datos, generalmente en una base de datos— que se desea distribuir bajo demanda a cierto conjunto de máquinas o personas. Una clave para comprender el concepto de cliente/servidor es que el depósito de información está ubicado centralmente, de manera que puede ser modificado y de forma que los cambios se propaguen a los consumidores de la información. A la(s) máquina(s) en las que se ubican conjuntamente el depósito de información y el software que la distribuye se la denomina el servidor. El software que reside en la máquina remota se comunica con el servidor, toma la información, la procesa y después la muestra en la máquina remota, denominada el *cliente*.

El concepto básico de la computación cliente/servidor, por tanto, no es tan complicado. Aparecen problemas porque se tiene un único servidor que trata de dar servicio a múltiples clientes simultáneamente. Generalmente, está involucrado algún sistema gestor de base de datos de manera que el diseñador "reparte" la capa de datos entre distintas tablas para lograr un uso óptimo de los mismos. Además, estos sistemas suelen admitir que un cliente inserte nueva información en el servidor. Esto significa que es necesario asegurarse de que el nuevo dato de un cliente no machaque los nuevos datos de otro cliente, o que no se pierda este dato en el proceso de su adición a la base de datos (a esto se le denomina procesamiento de la transacción). Al cambiar el software cliente, debe ser construido, depurado e instalado en las máquinas cliente, lo cual se vuelve bastante más complicado y caro de lo que pudiera parecer. Es especialmente problemático dar soporte a varios tipos de computadores y sistemas operativos. Finalmente, hay un aspecto de rendimiento muy importante: es posible tener cientos de clientes haciendo peticiones simultáneas a un mismo servidor, de forma que un mínimo retraso sea crucial. Para minimizar la latencia, los programadores deben empeñarse a fondo para disminuir las cargas de las tareas en proceso, generalmente repartiéndolas con las máquinas cliente, pero en ocasiones, se dirige la carga hacia otras máquinas ubicadas junto con el servidor, denominadas intermediarios "*middleware*"(que también se utiliza para mejorar la *mantenibilidad* del sistema global).

La simple idea de distribuir la información a la gente, tiene muchas capas de complejidad en la fase de implementación, y el problema como un todo puede parecer desesperanzador. E incluso puede ser crucial: la computación cliente/servidor se lleva prácticamente la mitad de todas las actividades de programación. Es responsable de todo, desde recibir las órdenes y transacciones de tarjetas de

crédito hasta la distribución de cualquier tipo de datos —mercado de valores, datos científicos, del gobierno,... Hasta la fecha, en el pasado, se han intentado y desarrollado soluciones individuales para problemas específicos, inventando una solución nueva cada vez. Estas soluciones eran difíciles de crear y utilizar, y el usuario debía aprenderse nuevas interfaces para cada una de ellas. El problema cliente/servidor completo debe resolverse con un enfoque global.

La Web como un servidor gigante

La Web es, de hecho, un sistema cliente/servidor gigante. Es un poco peor que eso, puesto que todos los servidores y clientes coexisten en una única red a la vez. No es necesario, sin embargo, ser conscientes de este hecho, puesto que simplemente es necesario preocuparse de saber cómo conectarse y cómo interactuar con un servidor en un momento dado (incluso aunque sea necesario merodear por todo el mundo para encontrar el servidor correcto).

Inicialmente, este proceso era unidireccional. Se hacía una petición de un servidor y éste te proporcionaba un archivo que el software navegador (por ejemplo, el cliente) de tu máquina podía interpretar dándole el formato adecuado en la máquina local. Pero en poco tiempo, la gente empezó a demandar más servicios que simplemente recibir páginas de un servidor. Se pedían capacidades cliente/servidor completas, de manera que el cliente pudiera retroalimentar de información al servidor; por ejemplo, para hacer búsquedas en base de datos en el servidor, añadir nueva información al mismo, o para ubicar una orden (lo que requería un nivel de seguridad mucho mayor que el que ofrecían los sistemas originales). Éstos son los cambios de los que hemos sido testigos a lo largo del desarrollo de la Web.

El navegador de la Web fue un gran paso hacia delante: el concepto de que un fragmento de información pudiera ser mostrado en cualquier tipo de computador sin necesidad de modificarlo. Sin embargo, los navegadores seguían siendo bastante primitivos y pronto se pasaron de moda debido a las demandas que se les hacían. No eran especialmente interactivos, y tendían a saturar tanto el servidor como Internet puesto que cada vez que requería hacer algo que exigiera programación había que enviar información de vuelta al servidor para que fuera procesada. Encontrar algo que por ejemplo, se había tecleado incorrectamente en una solicitud, podía llevar muchos minutos o segundos. Dado que el navegador era únicamente un visor no podía desempeñar ni siquiera las tareas de computación más simples. (Por otro lado, era seguro, puesto que no podía ejecutar programas en la máquina local que pudiera contener errores (*bugs*) o virus.)

Para resolver el problema, se han intentado distintos enfoques. El primero de ellos consistió en mejorar los estándares gráficos para permitir mejores animaciones y vídeos dentro de los navegadores. El resto del problema se puede resolver incorporando simplemente la capacidad de ejecutar programas en el cliente final, bajo el navegador. Esto se denomina programación en la parte cliente.

Programación en el lado del cliente

El diseño original servidor-navegador de la Web proporcionaba contenidos interactivos, pero la capacidad de interacción la proporcionaba completamente el servidor. Éste producía páginas estáticas para el navegador del cliente, que simplemente las interpretaba y visualizaba. El HTML básico

contiene mecanismos simples para la recopilación de datos: cajas de entrada de textos, cajas de prueba, cajas de radio, listas y listas desplegables, además de un botón que sólo podía programarse para borrar los datos del formulario o "enviar" los datos del formulario de vuelta al servidor. Este envío de datos se lleva a cabo a través del *Common Gateway Interface* (CGI), proporcionado por todos los servidores web. El texto del envío transmite a CGI qué debe hacer con él. La acción más común es ejecutar un programa localizado en el servidor en un directorio denominado generalmente "cgi-bin". (Si se echa un vistazo a la ventana de direcciones de la parte superior del navegador al presionar un botón de una página Web, es posible ver en ocasiones la cadena "cgi-bin" entre otras cosas.) Estos programas pueden escribirse en la mayoría de los lenguajes. Perl es una elección bastante frecuente pues fue diseñado para la manipulación de textos, y es interpretado, lo que permite que pueda ser instalado en cualquier servidor sin que importe el procesador o sistema operativo instalado.

Muchos de los sitios web importantes de hoy en día se siguen construyendo estrictamente con CGI, y es posible, de hecho, hacer casi cualquier cosa con él. Sin embargo, los sitios web cuyo funcionamiento se basa en programas CGI se suelen volver difíciles de mantener, y presentan además problemas de tiempo de respuesta. (Además, poner en marcha programas CGI suele ser bastante lento.) Los diseñadores iniciales de la Web no previeron la rapidez con que se agotaría el ancho de banda para los tipos de aplicaciones que se desarrollaron. Por ejemplo, es imposible llevar a cabo cualquier tipo de generación dinámica de gráficos con consistencia, pues es necesario crear un archivo GIF que pueda ser después trasladado del servidor al cliente para cada versión del gráfico. Y seguro que todo el mundo ha tenido alguna experiencia con algo tan simple como validar datos en un formulario de entrada. Se presiona el botón de enviar de una página; los datos se envían de vuelta al servidor; el servidor pone en marcha un programa CGI y descubre un error, da formato a una página HTML informando del error y después vuelve a mandar la página de vuelta; entonces es necesario recuperar el formulario y volver a empezar. Esto no es solamente lento, sino que es además poco elegante.

La solución es la programación en el lado del cliente. La mayoria de las máquinas que ejecutan navegadores Web son motores potentes capaces de llevar a cabo grandes cantidades de trabajo, y con el enfoque HTML estático original, simplemente estaban allí "sentadas", esperando ociosas a que el servidor se encargara de la página siguiente. La programación en el lado del cliente quiere decir que el servidor web tiene permiso para hacer cualquier trabajo del que sea capaz, y el resultado para el usuario es una experiencia mucho más rápida e interactiva en el sitio web.

El problema con las discusiones sobre la programación en el lado cliente es que no son muy distintas de las discusiones de programación en general. Los parámetros son casi los mismos, pero la plataforma es distinta: un navegador web es como un sistema operativo limitado. Al final, uno debe seguir programando, y esto hace que siga existiendo el clásico conjunto de problemas y soluciones, producidos en este caso por la programación en el lado del cliente. El resto de esta sección proporciona un repaso de los aspectos y enfoques en la programación en el lado del cliente.

Conectables *(plug-ins)*

Uno de los mayores avances en la programación en la parte cliente es el desarrollo de los conectables *(plug-ins)*. Éstos son modos en los que un programador puede añadir nueva funcionalidad al

navegador descargando fragmentos de código que se conecta en el punto adecuado del navegador. Le dice al navegador "de ahora en adelante eres capaz de llevar a cabo esta nueva actividad". (Es necesario descargar cada conectable únicamente una vez.) A través de los conectables, se añade comportamiento rápido y potente al navegador, pero la escritura de un conectable no es trivial y desde luego no es una parte deseable para hacer como parte de un proceso de construcción de un sitio web. El valor del conectable para la programación en el lado cliente es tal que permite a un programador experto desarrollar un nuevo lenguaje y añadirlo a un navegador sin permiso de la parte que desarrolló el propio navegador. Por consiguiente, los navegadores proporcionan una "puerta trasera que permite la creación de nuevos lenguajes de programación en el lado cliente (aunque no todos los lenguajes se encuentren implementados como conectables).

Lenguajes de guiones

Los conectables condujeron a la explosión de los lenguajes de guiones *(scripting)*. Con uno de estos lenguajes se integra el código fuente del programa de la parte cliente directamente en la página HTML, y el conectable que interpreta ese lenguaje se activa automáticamente a medida que se muestra la página HTML. Estos lenguajes tienden a ser bastante sencillos de entender y, dado que son simplemente texto que forma parte de una página HTML, se cargan muy rápidamente como parte del único acceso al servidor mediante el que se accede a esa página. El sacrificio es que todo el mundo puede ver (y robar) el código así transportado. Sin embargo, generalmente, si no se pretende hacer cosas excesivamente complicadas, los lenguajes de guiones constituyen una buena herramienta, al no ser complicados.

Esto muestra que los lenguajes de guiones utilizados dentro de los navegadores web se desarrollaron verdaderamente para resolver tipos de problemas específicos, en particular la creación de interfaces gráficos de usuario (IGUs) más interactivos y ricos. Sin embargo, uno de estos lenguajes puede resolver el 80% de los problemas que se presentan en la programación en el lado cliente. Este 80% podría además abarcar todos los problemas de muchos programadores, y dado que los lenguajes de programación permiten desarrollos mucho más sencillos y rápidos, es útil pensar en utilizar uno de estos lenguajes antes de buscar soluciones más complicadas, como la programación en Java o ActiveX.

Los lenguajes de guiones de navegador más comunes son JavaScript (que no tiene nada que ver con Java; se denominó así simplemente para aprovechar el buen momento de marketing de Java), VBScript (que se parece bastante a Visual Basic), y Tcl/Tk, que proviene del popular lenguaje de construcción de IGU (Interfaz Gráfica de Usuario) de plataformas cruzadas. Hay otros más, y seguro que se desarrollarán muchos más.

JavaScript es probablemente el que recibe más apoyo. Viene incorporado tanto en el navegador Netscape Navigator como en el Microsoft Internet Explorer (IE). Además, hay probablemente más libros de JavaScritpt que de otros lenguajes de navegador, y algunas herramientas crean páginas automáticamente haciendo uso de JavaScript. Sin embargo, si se tiene habilidad en el manejo de Visual Basic o Tcl/Tk, será más productivo hacer uso de esos lenguajes de guiones en vez de aprender uno nuevo. (De hecho, ya se habrá hecho uso de aspectos web para estas alturas.)

Java

Si un lenguaje de programación puede resolver el 80 por ciento de los problemas de programación en el lado cliente, ¿qué ocurre con el 20 por ciento restante —que constituyen de hecho "la

parte seria del problema"? La solución más popular hoy en día es Java. No sólo se trata de un lenguaje de programación potente, construido para ser seguro, de plataforma cruzada (multiplataforma) e internacional, sino que se está extendiendo continuamente para proporcionar aspectos de lenguaje y bibliotecas que manejan de manera elegante problemas que son complicados para los lenguajes de programación tradicionales, como la ejecución multihilo, el acceso a base de datos, la programación en red, y la computación distribuida. Java permite programación en el lado cliente a través del *applet*.

Un *applet* es un miniprograma que se ejecutará únicamente bajo un navegador web. El *applet* se descarga automáticamente como parte de una página web (igual que, por ejemplo, se descarga un gráfico, de manera automática). Cuando se activa un *applet*, ejecuta un programa. Ésta es parte de su belleza —proporciona una manera de distribuir automáticamente software cliente desde el servidor justo cuando el usuario necesita software cliente, y no antes. El usuario se hace con la última versión del software cliente, sin posibilidad de fallo, y sin tener que llevar a cabo reinstalaciones complicadas. Gracias a cómo se ha diseñado Java, el programado simplemente tiene que crear un único programa, y este programa trabaja automáticamente en todos los computadores que tengan navegadores que incluyan intérpretes de Java. (Esto incluye seguramente a la gran mayoría de plataformas.) Dado que Java es un lenguaje de programación para novatos, es posible hacer tanto trabajo como sea posible en el cliente, antes y después de hacer peticiones al servidor. Por ejemplo, no se deseará enviar un formulario de petición a través de Internet para descubrir que se tiene una fecha o algún otro parámetro erróneo, y el computador cliente puede llevar a cabo rápidamente la labor de registrar información en vez de tener que esperar a que lo haga el servidor para enviar después una imagen gráfica de vuelta. No sólo se consigue un incremento de velocidad y capacidad de respuesta inmediatas, sino que el tráfico en general de la red y la carga en los servidores se reduce considerablemente, evitando que toda Internet se vaya ralentizando.

Una ventaja que tienen los *applets* de Java sobre los lenguajes de guiones es que se encuentra en formato compilado, de forma que el código fuente no está disponible para el cliente. Por otro lado, es posible descompilar un *applet* Java sin excesivo trabajo, pero esconder un código no es un problema generalmente importante. Hay otros dos factores que pueden ser importantes. Como se verá más tarde en este libro, un *applet* Java compilado puede incluir varios módulos e implicar varios accesos al servidor para su descarga (en Java 1.1 y superiores, esto se minimiza mediante archivos Java, denominados archivos JAR, que permiten que los módulos se empaqueten todos juntos y se compriman después para que baste con una única descarga). Un programa de guiones se integrará simplemente en una página web como parte de su texto (y generalmente será más pequeño reduciendo los accesos al servidor). Esto podría ser importante de cara al tiempo de respuesta del sitio web. Otro factor importante es la curva de aprendizaje. A pesar de lo que haya podido oírse, Java no es un lenguaje cuyo aprendizaje resulte trivial. Para los programadores en Visual Basic, moverse a VBScript siempre será la solución más rápida, y dado que probablemente este lenguaje resolverá los problemas cliente/servidor más típicos, puede resultar bastante complicado justificar la necesidad de aprender Java. Si uno ya tiene experiencia con un lenguaje de guiones, seguro que se obtendrán beneficios simplemente haciendo uso de JavaScript o VBScript antes de lanzarse a utilizar Java, ya que estos lenguajes pueden resolver todos los problemas de manera sencilla, y se logrará un nivel de productividad elevado en un tiempo menor.

ActiveX

Hasta cierto grado, el competidor principal de Java es el ActiveX de Microsoft, aunque se base en un enfoque totalmente diferente. ActiveX era originalmente una solución válida exclusivamente para Windows, aunque ahora se está desarrollando mediante un consorcio independiente de manera que acabará siendo multiplataforma (plataforma cruzada). Efectivamente, ActiveX se basa en que "si un programa se conecta a su entorno de manera que puede ser depositado en una página web y ejecutado en un navegador, entonces soporta ActiveX". (IE soporta ActiveX directamente, y Netscape también, haciendo uso de un conectable.) Por consiguiente, ActiveX no se limita a un lenguaje particular. Si, por ejemplo, uno es un programador Windows experimentado, haciendo uso de un lenguaje como C++, Visual Basic o en Delphi de Borland, es posible crear componentes ActiveX sin casi tener que hacer ningún cambio a los conocimientos de programación que ya se tengan. Además, ActiveX proporciona un modo de usar código antiguo (base dado) en páginas web.

Seguridad

La capacidad para descargar y ejecutar programas a través de Internet puede parecer el sueño de un constructor de virus. ActiveX atrae especialmente el espinoso tema de la seguridad en la programación en la parte cliente. Si se hace *clic* en el sitio web, es posible descargar automáticamente cualquier número de cosas junto con la página HTML: archivos GIF, código de guiones, código Java compilado, y componentes ActiveX. Algunos de estos elementos son benignos: los archivos GIF no pueden hacer ningún daño, y los lenguajes de guiones se encuentran generalmente limitados en lo que pueden hacer. Java también fue diseñado para ejecutar sus *applets* dentro de un "envoltorio" de seguridad, lo que evita que escriba en el disco o acceda a la memoria externa a ese envoltorio.

ActiveX está en el rango opuesto del espectro. Programar con ActiveX es como programar Windows —es posible hacer cualquier cosa. De esta manera, si se hace *clic* en una página web que descarga un componente ActiveX, ese componente podría llegar a dañar los archivos de un disco. Por supuesto, los programas que uno carga en un computador, y que no están restringidos a ejecutarse dentro del navegador web podrían hacer lo mismo. Los virus que se descargaban desde una *BBS (Bulletin-Board Systems)* hace ya tiempo que son un problema, pero la velocidad de Internet amplifica su gravedad.

La solución parecen aportarla las "firmas digitales", que permiten la verificación de la autoría del código. Este sistema se basa en la idea de que un virus funciona porque su creador puede ser anónimo, de manera que si se evita la ejecución de programas anónimos, se obligará a cada persona a ser responsable de sus actos. Esto parece una buena idea, pues permite a los programas ser mucho más funcionales, y sospecho que eliminará las diabluras maliciosas. Si, sin embargo, un programa tiene un error (*bug*) inintencionadamente destructivo, seguirá causando problemas.

El enfoque de Java es prevenir que estos problemas ocurran, a través del envoltorio. El intérprete de Java que reside en el navegador web local examina el *applet* buscando instrucciones adversas a medida que se carga el *applet*. Más en concreto, el *applet* no puede escribir ficheros en el disco o borrar ficheros (una de las principales vías de ataque de los virus). Los *applets* se consideran generalmente seguros, y dado que esto es esencial para lograr sistemas cliente/servidor de confianza, cualquier error (*bug*) que produzca virus en lenguaje Java será rápidamente reparado. (Merece la

pena destacar que el software navegador, de hecho, refuerza estas restricciones de seguridad, y algunos navegadores permiten seleccionar distintos niveles de seguridad para proporcionar distintos grados de acceso a un sistema.)

También podría uno ser escéptico sobre esta restricción tan draconiana en contra de la escritura de ficheros en un disco local. Por ejemplo, uno puede desear construir una base de datos o almacenar datos para utilizarlos posteriormente, finalizada la conexión. La visión inicial parecía ser tal que eventualmente todo el mundo podría conseguir hacer cualquier cosa importante estando conectado, pero pronto se vio que esta visión no era práctica (aunque los "elementos Internet" de bajo coste puedan satisfacer algún día las necesidades de un segmento de usuarios significativo). La solución es el "*applet* firmado" que utiliza cifrado de clave pública para verificar que un *applet* viene efectivamente de donde dice venir. Un *applet* firmado puede seguir destrozando un disco local, pero la teoría es que dado que ahora es posible localizar al creador del *applet*, éstos no actuarán de manera perniciosa. Java proporciona un marco de trabajo para las firmas digitales, de forma que será posible permitir que un *applet* llegue a salir fuera del envoltorio si es necesario.

Las firmas digitales han olvidado un aspecto importante, que es la velocidad con la que la gente se mueve por Internet. Si se descarga un programa con errores (*bugs*) que hace algo dañino, ¿cuánto tiempo se tardará en descubrir el daño? Podrían pasar días o incluso semanas. Para entonces, ¿cómo localizar el programa que lo ha causado? ¿Y será todo el mundo capaz de hacerlo?

Internet frente a Intranet

La Web es la solución más general al problema cliente/servidor, de forma que tiene sentido que se pueda utilizar la misma tecnología para resolver un subconjunto del problema, en particular, el clásico problema cliente/servidor *dentro* de una compañía. Con los enfoques cliente/servidor tradicionales, se tiene el problema de la multiplicidad de tipos de máquinas cliente, además de la dificultad de instalar un nuevo software cliente, si bien ambos problemas pueden resolverse sencillamente con navegadores web y programación en el lado cliente. Cuando se utiliza tecnología web para una red de información restringida a una compañía en particular, se la denomina una Intranet. Las Intranets proporcionan un nivel de seguridad mucho mayor que el de Internet, puesto que se puede controlar físicamente el acceso a los servidores dentro de la propia compañía. En términos de formación, parece que una vez que la gente entiende el concepto general de navegador es mucho más sencillo que se enfrenten a distintas páginas y *applets*, de manera que la curva de aprendizaje para nuevos tipos de sistemas parece reducirse.

El problema de la seguridad nos conduce ante una de las divisiones que parece estar formándose automáticamente en el mundo de la programación en el lado del cliente. Si un programa se ejecuta en Internet, no se sabe bajo qué plataforma estará funcionando, y si se desea ser extremadamente cauto, no se diseminará código con error. Es necesario algo multiplataforma y seguro, como un lenguaje de guiones o Java.

Si se está ejecutando código en una Intranet, es posible tener un conjunto de limitaciones distinto. No es extraño que las máquinas de una red puedan ser todas plataformas Intel/Windows. En una Intranet, uno es responsable de la calidad de su propio código y puede reparar errores en el momento en que se descubren. Además, se podría tener cierta cantidad de código antiguo (heredado, *legacy*) que se ha

estado utilizando en un enfoque cliente/servidor más tradicional, en cuyo caso sería necesario instalar físicamente programas cliente cada vez que se construya una versión más moderna. El tiempo malgastado en instalar actualizaciones (*upgrades*) es la razón más apabullante para comenzar a usar navegadores, en los que estas actualizaciones son invisibles y automáticas. Para aquéllos que tengan intranets, el enfoque más sensato es tomar el camino más corto que permita usar el código base existente, en vez de volver a codificar todos los programas en un nuevo lenguaje.

Se ha hecho frente a este problema presentando un conjunto desconcertante de soluciones al problema de programación en el lado cliente, y la mejor determinación para cada caso es la que determine un análisis coste-beneficio. Deben considerarse las restricciones de cada problema y cuál sería el camino más corto para encontrar la solución en cada caso. Dado que la programación en la parte cliente sigue siendo programación, suele ser buena idea tomar el enfoque de desarrollo más rápido para cada situación. Ésta es una postura agresiva para prepararse de cara a inevitables enfrentamientos con los problemas del desarrollo de programas.

Programación en el lado del servidor

Hasta la fecha toda discusión ha ignorado el problema de la programación en el lado del servidor. ¿Qué ocurre cuando se hace una petición a un servidor? La mayoría de las veces la petición es simplemente "envíame este archivo". A continuación, el navegador interpreta el archivo de la manera adecuada: como una página HTML, como una imagen gráfica, un *applet* Java, un programa de guiones, etc. Una petición más complicada hecha a un servidor puede involucrar una transacción de base de datos. Un escenario común involucra una petición para una búsqueda compleja en una base de datos, que el servidor formatea en una página HTML para enviarla a modo de resultado (por supuesto, si el cliente tiene una inteligencia mayor vía Java o un lenguaje de guiones, pueden enviarse los datos simplemente, sin formato, y será el extremo cliente el que les dé el formato adecuado, lo cual es más rápido, además de implicar una carga menor para el servidor). Un usuario también podría querer registrar su nombre en una base de datos al incorporarse a un grupo o presentar una orden, lo cual implica cambios a esa base de datos. Estas peticiones deben procesarse vía algún código en el lado servidor, que se denomina generalmente programación en el lado servidor. Tradicionalmente, esta programación se ha desempeñado mediante Perl y guiones CGI, pero han ido apareciendo sistemas más sofisticados. Entre éstos se encuentran los servidores web basados en Java que permiten llevar a cabo toda la programación del lado servidor en Java escribiendo lo que se denominan *servlets*. Éstos y sus descendientes, los JSP, son las dos razones principales por las que las compañías que desarrollan sitios web se están pasando a Java, especialmente porque eliminan los problemas de tener que tratar con navegadores de distintas características.

Un ruedo separado: las aplicaciones

Muchos de los comentarios en torno a Java se referían a los *applets*. Java es actualmente un lenguaje de programación de propósito general que puede resolver cualquier tipo de problema —al menos en teoría. Y como se ha señalado anteriormente, cuando uno se sale del ruedo de los *applets* (y simultáneamente se salta las restricciones, como la contraria a la escritura en el disco) se entra en el mundo de las aplicaciones de propósito general que se ejecutan independientemente, sin un nave-

gador web, al igual que hace cualquier programa ordinario. Aquí, la fuerza de Java no es sólo su portabilidad, sino también su programabilidad (facilidad de programación). Como se verá a lo largo del presente libro, Java tiene muchos aspectos que permiten la creación de programas robustos en un período de tiempo menor al que requerían los lenguajes de programación anteriores.

Uno debe ser consciente de que esta bendición no lo es del todo. El precio a pagar por todas estas mejoras es una velocidad de ejecución menor (aunque se está haciendo bastante trabajo en este área —JDK 1.3, en particular, presenta las mejoras de rendimiento denominadas "*hotspot*"). Como cualquier lenguaje, Java tiene limitaciones intrínsecas que podrían hacerlo inadecuado para resolver cierto tipo de problemas de programación. Java es un lenguaje que evoluciona rápidamente, no obstante, y cada vez que aparece una nueva versión, se presenta más y más atractivo de cara a la solución de conjuntos mayores de problemas.

Análisis y diseño

El paradigma de la orientación a objetos es una nueva manera de enfocar la programación. Son muchos los que tienen problemas a primera vista para enfrentarse a un proyecto de POO. Dado que se supone que todo es un objeto, y a medida que se aprende a pensar de forma orientada a objetos, es posible empezar a crear "buenos" diseños y sacar ventaja de todos los beneficios que la POO puede ofrecer.

Una *metodología* es un conjunto de procesos y heurísticas utilizadas para descomponer la complejidad de un problema de programación. Se han formulado muchos métodos de POO desde que enunció la programación orientada a objetos. Esta sección presenta una idea de lo que se trata de lograr al utilizar un método.

Especialmente en la POO, la metodología es un área de intensa experimentación, por lo que es importante entender qué problema está intentando resolver el método antes de considerar la adopción de uno de ellos. Esto es particularmente cierto con Java, donde el lenguaje de programación se ha desarrollado para reducir la complejidad (en comparación con C) necesaria para expresar un programa. Esto puede, de hecho, aliviar la necesidad de metodologías cada vez más complejas. En vez de esto, puede que las metodologías simples sean suficientes en Java para conjuntos de problemas mucho mayores que los que se podrían manipular utilizando metodologías simples con lenguajes procedimentales.

También es importante darse cuenta de que el término "metodología" es a menudo muy general y promete demasiado. Se haga lo que se haga al diseñar y escribir un programa, se sigue un método. Puede que sea el método propio de uno, e incluso puede que uno no sea consciente de utilizarlo, pero es un proceso que se sigue al crear un programa. Si el proceso es efectivo, puede que simplemente sea necesario afinarlo ligeramente para poder trabajar con Java. Si no se está satisfecho con el nivel de productividad y la manera en que se comportan los programas, puede ser buena idea considerar la adopción de un método formal, o la selección de fragmentos de entre los muchos métodos formales existentes.

Mientras se está en el propio proceso de desarrollo, el aspecto más importante es no perderse, aunque puede resultar fácil. Muchos de los métodos de análisis y desarrollo fueron concebidos para re-

solver los problemas más grandes. Hay que recordar que la mayoría de proyectos no encajan en esta categoría, siendo posible muchas veces lograr un análisis y un diseño con éxito con sólo un pequeño subconjunto de los que el método recomienda[10]. Pero algunos tipos de procesos, sin importar lo limitados que puedan ser, le permitirán encontrar el camino de manera más sencilla que si simplemente se empieza a codificar.

También es fácil desesperarse, caer en "parálisis de análisis", cuando se siente que no se puede avanzar porque no se han cubierto todos los detalles en la etapa actual. Debe recordarse que, independientemente de cuánto análisis lleve a cabo, hay cosas de un sistema que no aparecerán hasta la fase de diseño, y otras no aflorarán incluso hasta la fase de codificación o en un extremo, hasta que el programa esté acabado y en ejecución. Debido a esto, es crucial moverse lo suficientemente rápido a través de las etapas de análisis y diseño e implementar un prototipo del sistema propuesto.

Debe prestarse un especial énfasis a este punto. Dado que ya se conoce la misma historia con los lenguajes *procedimentales*, es recomendable que el equipo proceda de manera cuidadosa y comprenda cada detalle antes de pasar del diseño a la implementación. Ciertamente, al crear un SGBD, esto pasa por comprender completamente la necesidad del cliente. Pero un SGBD es la clase de problema bien formulada y bien entendida; en muchos programas, *es* la estructura de la base de datos la que debe ser desmenuzada. La clase de problema de programación examinada en el presente capítulo es un "juego de azar"[11], en el que la solución no es simplemente la formulación de una solución bien conocida, sino que involucra además a uno o más "factores de azar" —elementos para los que no existe una solución previa bien entendida, y para los cuales es necesario algún tipo de proceso de investigación[12]. Intentar analizar completamente un problema al azar antes de pasar al diseño e implementación conduce a una parálisis en el análisis, al no tener suficiente información para resolver este tipo de problemas durante la fase de análisis. Resolver un problema así, requiere iterar todo el ciclo, y precisa de un comportamiento que asuma riesgos (lo cual tiene sentido, pues está intentando hacer algo nuevo y las recompensas potenciales crecen). Puede parecer que el riesgo se agrava al precipitarse hacia una implementación preliminar, pero ésta puede reducir el riesgo en los problemas al azar porque se está averiguando muy pronto si un enfoque particular al problema es o no viable. El desarrollo de productos conlleva una gestión del riesgo.

A menudo, se propone "construir uno para desecharlo". Con POO es posible tirar *parte*, pero dado que el código está encapsulado en clases, durante la primera pasada siempre se producirá algún diseño de clases útil y se desarrollarán ideas que merezcan la pena para el diseño del sistema de las que no habrá que deshacerse. Por tanto, la primera pasada rápida por un problema no sólo suministra información crítica para las ulteriores pasadas por análisis, diseño e implementación, sino que también crea la base del código.

---

[10] Un ejemplo excelente de esto es *UML Distilled*, 2.ª edición, de Martin Fowler (Addison-Wesley 2000), que reduce el proceso, en ocasiones aplastante, a un subconjunto manejable (existe versión española con el título *UML gota a gota*).

[11] N. del traductor: Término *wild-card*, acuñado por el autor original.

[12] Regla del pulgar —acuñada por el autor— para estimar este tipo de proyectos: si hay más de un factor al azar, ni siquiera debe intentarse planificar la duración o el coste del proyecto hasta que no se ha creado un prototipo que funcione. Existen demasiados grados de libertad.

Dicho esto, si se está buscando una metodología que contenga un nivel de detalle tremendo, y sugiera muchos pasos y documentos, puede seguir siendo difícil saber cuándo parar. Debe recomendarse lo que se está intentando descubrir.

1. ¿Cuáles son los objetos? (¿Cómo se descompone su proyecto en sus componentes?)

2. ¿Cuáles son las interfaces? (¿Qué mensajes es necesario enviar a cada objeto?)

Si se delimitan los objetos y sus interfaces, ya es posible escribir un programa. Por diversas razones, puede que sean necesarias más descripciones y documentos que éste, pero no es posible avanzar con menos.

El proceso puede descomponerse en cinco fases, y la Fase 0 no es más que la adopción de un compromiso para utilizar algún tipo de estructura.

Fase 0: Elaborar un plan

En primer lugar, debe decidirse qué pasos debe haber en un determinado proceso. Suena bastante simple (de hecho, *todo* esto suena simple) y la gente no obstante, suele seguir sin tomar esta decisión antes de empezar a codificar. Si el plan consiste en "empecemos codificando", entonces, perfecto (en ocasiones, esto es apropiado, si uno se está enfrentando a un problema que conoce perfectamente). Al menos, hay que estar de acuerdo en que eso también es tener un plan.

También podría decidirse en esta fase que es necesaria alguna estructura adicional de proceso, pero no toda una metodología completa. Para que nos entendamos, a algunos programadores les gusta trabajar en "modo vacación", en el que no se imponga ninguna estructura en el proceso de desarrollar de su trabajo; "se hará cuando se haga". Esto puede resultar atractivo a primera vista, pero a medida que se tiene algo de experiencia uno se da cuenta de que es mejor ordenar y distribuir el esfuerzo en distintas etapas en vez de lanzarse directamente a "finalizar el proyecto". Además, de esta manera se divide el proyecto en fragmentos más asequibles, y se resta miedo a la tarea de enfrentarse al mismo (además, las distintas fases o hitos proporcionan más motivos de celebración).

Cuando empecé a estudiar la estructura de la historia (con el propósito de acabar escribiendo algún día una novela), inicialmente, la idea que más me disgustaba era la de la estructura, pues parecía que uno escribe mejor si simplemente se dedica a rellenar páginas. Pero más tarde descubrí que al escribir sobre computadores, tenía la estructura tan clara que no había que pensar demasiado en ella. Pero aún así, el trabajo se estructuraba, aunque sólo fuera semiconscientemente en mi cabeza. Incluso cuando uno piensa que el plan consiste simplemente en empezar a codificar, todavía se atraviesan algunas fases al plantear y contestar ciertas preguntas.

El enunciado de la misión

Cualquier sistema que uno construya, independientemente de lo complicado que sea, tiene un propósito fundamental: el negocio intrínseco en el mismo, la necesidad básica que cumple. Si uno puede mirar a través de la interfaz de usuario, a los detalles específicos del hardware o del sistema, los algoritmos de codificación y los problemas de eficiencia, entonces se encuentra el centro de su existencia —simple y directo. Como el denominado alto concepto (*high concept*) en las películas de

Hollywood, uno puede describir el propósito de un programa en dos o tres fases. Esta descripción, pura, es el punto de partida.

El alto concepto es bastante importante porque establece el tono del proyecto; es el enunciado de su misión. Uno no tiene por qué acertar necesariamente a la primera (puede ser que uno esté en una fase posterior del problema cuando se le ocurra el enunciado completamente correcto) pero hay que seguir intentándolo hasta tener la certeza de que está bien. Por ejemplo, en un sistema de control de tráfico aéreo, uno puede comenzar con un alto concepto centrado en el sistema que se está construyendo: "El programa de la torre hace un seguimiento del avión". Pero considérese qué ocurre cuando se introduce el sistema en un pequeño aeródromo; quizás sólo hay un controlador humano, o incluso ninguno. Un modelo más usual no abordará la solución que se está creando como describe el problema: "Los aviones llegan, descargan, son mantenidos y recargan, a continuación, salen".

Fase 1: ¿Qué estamos construyendo?

En la generación previa del diseño del programa (denominada *diseño procedural*) a esta fase se le denominaba "creación del *análisis de requisitos* y *especificación del sistema*". Éstas, por supuesto, eran fases en las que uno se perdía; documentos con nombres intimidadores que podían de por sí convertirse en grandes proyectos. Sin embargo, su intención era buena. El análisis de requisitos dice: "Construya una lista de directrices que se utilizarán para saber cuándo se ha acabado el trabajo y cuándo el cliente está satisfecho". La especificación del sistema dice: "He aquí una descripción de lo *que* el programa hará (pero no *cómo*) para satisfacer los requisitos hallados". El análisis de requisitos es verdaderamente un contrato entre usted y el cliente (incluso si el cliente trabaja en la misma compañía o es cualquier otro objeto o sistema). La especificación del sistema es una exploración de alto nivel en el problema, y en cierta medida, un descubrimiento de si puede hacerse y cuánto tiempo llevará. Dado que ambos requieren de consenso entre la gente (y dado que generalmente variarán a lo largo del tiempo) lo mejor es mantenerlos lo más desnudos posible —idealmente, tratará de listas y diagramas básicos para ahorrar tiempo. Se podría tener otras limitaciones que exijan expandirlos en documentos de mayor tamaño, pero si se mantiene que el documento inicial sea pequeño y conciso, es posible crearlo en unas pocas sesiones de tormenta de ideas (*brainstorming*) en grupo, con un líder que dinámicamente va creando la descripción. Este proceso no sólo exige que todos aporten sus ideas sino que fomenta el que todos los miembros del equipo lleguen a un acuerdo inicial. Quizás lo más importante es que puede incluso ayudar a que se acometa el proyecto con una gran dosis de entusiasmo.

Es necesario mantenerse centrado en el corazón de lo que se está intentando acometer en esta fase: determinar qué es lo que se supone que debe hacer el sistema. La herramienta más valiosa para esto es una colección de lo que se denomina "casos de uso". Los casos de uso identifican los aspectos claves del sistema, que acabarán por revelar las clases fundamentales que se usarán en éste. De hecho, los casos de uso son esencialmente soluciones descriptivas a preguntas como[13]:

- "¿Quién usará el sistema?"
- "¿Qué pueden hacer esos actores con el sistema?"

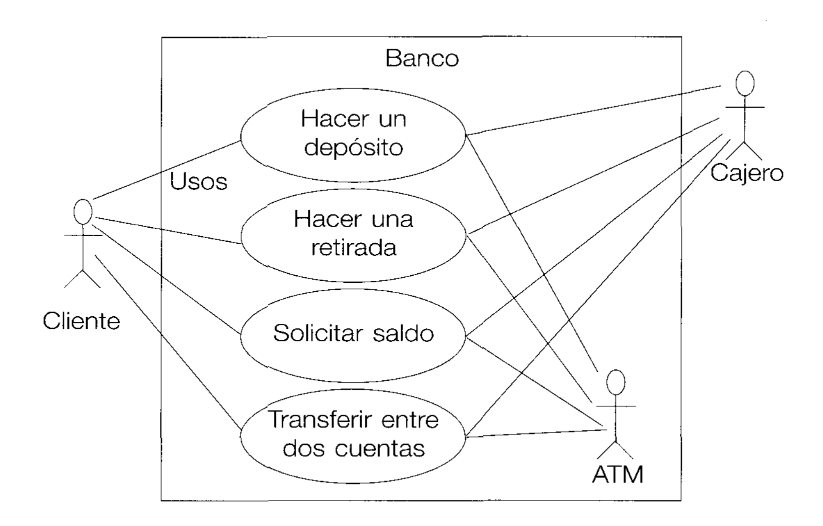
---

[13] Agradecemos la ayuda de James H. Jarrett.

- ¿Cómo se las ingenia *cada actor* para *hacer* eso con este sistema?"
- "¿De qué otra forma podría funcionar esto si alguien más lo estuviera haciendo, o si el mismo actor tuviera un objetivo distinto?" (Para encontrar posibles variaciones.)
- "¿Qué problemas podrían surgir mientras se hace esto con el sistema?" (Para localizar posibles excepciones.)

Si se está diseñando, por ejemplo, un cajero automático, el caso de uso para un aspecto particular de la funcionalidad del sistema debe ser capaz de describir qué hace el cajero en cada situación posible. Cada una de estas "situaciones" se denomina un *escenario*, y un caso de uso puede considerarse como una colección de escenarios. Uno puede pensar que un escenario es como una pregunta que empieza por: "¿Qué hace el sistema si...?". Por ejemplo: "¿Qué hace el cajero si un cliente acaba de depositar durante las últimas 24 horas un cheque y no hay dinero suficiente en la cuenta, sin haber procesado el cheque, para proporcionarle la retirada el efectivo que ha solicitado?"

Deben utilizarse diagramas de caso de uso intencionadamente simples para evitar ahogarse prematuramente en detalles de implementación del sistema :

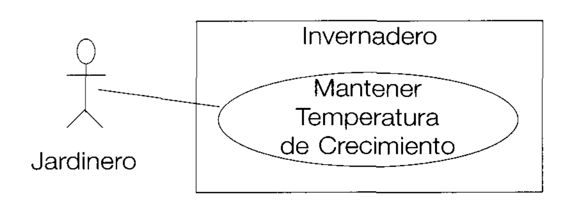


Cada uno de los monigotes representa a un "actor", que suele ser generalmente un humano o cualquier otro tipo de agente (por ejemplo, otro sistema de computación, como "ATM")[14]. La caja representa los límites de nuestro sistema. Las elipses representan los casos de uso, que son descripciones del trabajo útil que puede hacerse dentro del sistema. Las líneas entre los actores y los casos de uso representan las interacciones.

De hecho no importa cómo esté el sistema implementado, siempre y cuando tenga una apariencia como ésta para el usuario.

[14] ATM, siglas en inglés de cajero automático. (N. del T.)

Un caso de uso no tiene por qué ser terriblemente complejo, aunque el sistema subyacente sea complejo. Solamente se pretende que muestre el sistema tal y como éste se muestra al usuario. Por ejemplo:



Los casos de uso proporcionan las especificaciones de requisitos determinando todas las interacciones que el usuario podría tener con el sistema. Se trata de descubrir un conjunto completo de casos de uso para su sistema, y una vez hecho esto, se tiene el núcleo de lo que el sistema se supone que hará. Lo mejor de centrarse en los casos de uso es que siempre permiten volver a la esencia manteniéndose alejado de aspectos que no son críticos para lograr culminar el trabajo. Es decir, si se tiene un conjunto completo de casos de uso, es posible describir el sistema y pasar a la siguiente fase. Posiblemente no se podrá configurar todo a la primera, pero no pasa nada. Todo irá surgiendo a su tiempo, y si se demanda una especificación perfecta del sistema en este punto, uno se quedará parado.

Cuando uno se quede bloqueado, es posible comenzar esta fase utilizando una extensa herramienta de aproximación: describir el sistema en unos pocos párrafos y después localizar los sustantivos y los verbos. Los sustantivos pueden sugerir quiénes son los actores, el contexto del caso de uso (por ejemplo, "corredor"), o artefactos manipulados en el caso de uso. Los verbos pueden sugerir interacciones entre los actores y los casos de uso, y especificar los pasos dentro del caso de uso. También será posible descubrir que los sustantivos y los verbos producen objetos y mensajes durante la fase de diseño (y debe tenerse en cuenta que los casos de uso describen interacciones entre subsistemas, de forma que la técnica de "el sustantivo y el verbo" puede usarse sólo como una herramienta de tormenta de ideas, pues no genera casos de uso)[15].

La frontera entre un caso de uso y un actor puede señalar la existencia de una interfaz de usuario, pero no lo define. Para ver el proceso de cómo definir y crear interfaces de usuario, véase *Software for Use* de Larry Constantine y Lucy Lockwood, (Addison-Wesley Longman, 1999) o ir a *http://www.ForUse.com.*

Aunque parezca magia negra, en este punto es necesario algún tipo de planificación. Ahora se tiene una visión de lo que se está construyendo, por lo que probablemente se pueda tener una idea de cuánto tiempo le llevará. En este momento intervienen muchos factores. Si se estima una planificación larga, la compañía puede decidir no construirlo (y por consiguiente usar sus recursos en algo más razonable —esto es *bueno*). Pero un director podría tener decidido de antemano cuánto tiempo debería llevar el proyecto y podría tratar de influir en la estimación. Pero lo mejor es tener una estimación honesta desde el principio y tratar las decisiones duras al principio. Ha habido muchos in-

[15] Puede encontrarse más información sobre casos de uso en *Applying Use Cases*, de Schneider & Winters (Addison-Weley 1998) y *Use Case Driven Object modeling with UML* de Rosenberg (Addison-Welsey 1999).

tentos de desarrollar técnicas de planificación exactas (muy parecidas a las técnicas de predicción del mercado de valores), pero probablemente el mejor enfoque es confiar en la experiencia e intuición. Debería empezarse por una primera estimación del tiempo que llevaría, para posteriormente multiplicarla por dos y añadirle el 10 por ciento. La estimación inicial puede que sea correcta; a lo mejor *se puede* hacer que algo funcione en ese tiempo. Al "doblarlo" resultará que se consigue algo decente, y en el 10 por ciento añadido se puede acabar de pulir y tratar los detalles finales[16]. Sin embargo, es necesario explicarlo, y dejando de lado las manipulaciones y quejas que surgen al presentar una planificación de este tipo, normalmente funcionará.

Fase 2: ¿Cómo construirlo?

En esta fase debe lograrse un diseño que describe cómo son las clases y cómo interactuarán. Una técnica excelente para determinar las clases e interacciones es la tarjeta *Clase-Responsabilidad-Colaboración* (CRC)[17]. Parte del valor de esta herramienta se basa en que es de muy baja tecnología: se comienza con un conjunto de tarjetas de 3 x 5, y se escribe en ellas. Cada tarjeta representa una única clase, y en ella se escribe:

1. El nombre de la clase. Es importante que este nombre capture la esencia de lo que hace la clase, de manera que tenga sentido a primera vista.

2. Las "responsabilidades" de la clase: qué debería hacer. Esto puede resumirse típicamente escribiendo simplemente los nombres de las funciones miembros (dado que esas funciones deberían ser descriptivas en un buen diseño), pero no excluye otras anotaciones. Si se necesita ayuda, basta con mirar el problema desde el punto de vista de un programador holgazán: ¿qué objetos te gustaría que apareciesen por arte de magia para resolver el problema?

3. Las "colaboraciones" de la clase: ¿con qué otras clases interactúa? "Interactuar" es un término amplio intencionadamente; vendría a significar agregación, o simplemente que cualquier otro objeto existente ejecutara servicios para un objeto de la clase. Las colaboraciones deberían considerar también la audiencia de esa clase. Por ejemplo, si se crea una clase **Petardo**, ¿quién la va a observar, un **Químico** o un **Observador**? En el primer caso estamos hablando de punto de vista del químico que va a construirlo, mientras que en el segundo se hace referencia a los colores y las formas que libere al explotar.

Uno puede pensar que las tarjetas deberían ser más grandes para que cupiera en ellas toda la información que se deseara escribir, pero son pequeñas a propósito, no sólo para mantener pequeño el tamaño de las clases, sino también para evitar que se caiga en demasiado nivel de detalle muy pronto. Si uno no puede encajar todo lo que necesita saber de una clase en una pequeña tarjeta, la clase es demasiado compleja (o se está entrando en demasiado nivel de detalle, o se debería crear más de una clase). La clase ideal debería ser comprensible a primera vista. La idea de las tarjetas CRC es ayudar a obtener un primer diseño de manera que se tenga un dibujo a grandes rasgos que pueda ser después refinado.

[16] Mi opinión en este sentido ha cambiado últimamente. Al doblar y añadir el 10 por ciento se obtiene una estimación razonablemente exacta (asumiendo que no hay demasiados factores al azar) pero todavía hay que trabajar con bastante diligencia para finalizar en ese tiempo. Si se desea tener tiempo suficiente para lograr un producto verdaderamente elegante y disfrutar durante el proceso, el multiplicador correcto, en mi opinión, puede ser por tres o por cuatro.

[17] En inglés, *Class-Responsibility-Collaboration*. (N. del R.T.)

Una de las mayores ventajas de las tarjetas CRC se logra en la comunicación. Cuando mejor se hace es en tiempo real, en grupo y sin computadores. Cada persona se considera responsable de varias clases (que al principio no tienen ni nombres ni otras informaciones). Se ejecuta una simulación en directo resolviendo cada vez un escenario, decidiendo qué mensajes se mandan a los distintos objetos para satisfacer cada escenario. A medida que se averiguan las responsabilidades y colaboraciones de cada una, se van rellenando las tarjetas correspondientes. Cuando se han recorrido todos los casos de uso, se debería tener un diseño bastante completo.

Antes de empezar a usar tarjetas CRC, tuve una experiencia de consultoría de gran éxito, que me permitió presentar un diseño inicial a todo el equipo, que jamás había participado en un proyecto de POO, y que consistió en ir dibujando objetos en una pizarra, después de hablar sobre cómo se deberían comunicar los objetos entre sí, y tras borrar algunos y reemplazar otros. Efectivamente, estaban haciendo uso de "tarjetas CRC" en la propia pizarra. El equipo (que sabía que el proyecto se iba a hacer) creó, de hecho, el diseño; ellos eran los "propietarios" del diseño, más que recibirlo hecho directamente. Todo lo que yo hacía era guiar el proceso haciendo en cada momento las preguntas adecuadas, poniendo a prueba los distintos supuestos, y tomando la realimentación del equipo para ir modificando los supuestos. La verdadera belleza del proyecto es que el equipo aprendió cómo hacer diseño orientado a objetos no repasando ejemplos o resúmenes de ejemplos, sino trabajando en el diseño que les pareció más interesante en ese momento: el de ellos mismos.

Una vez que se tiene un conjunto de tarjetas CRC se desea crear una descripción más formal del diseño haciendo uso de UML[18]. No es necesario utilizar UML, pero puede ser de gran ayuda, especialmente si se desea poner un diagrama en la pared para que todo el mundo pueda ponderarlo, lo cual es una gran idea. Una alternativa a UML es una descripción textual de los objetos y sus interfaces, o, dependiendo del lenguaje de programación, el propio código[19].

UML también proporciona una notación para diagramas que permiten describir el modelo dinámico del sistema. Esto es útil en situaciones en las que las transiciones de estado de un sistema o subsistema son lo suficientemente dominantes como para necesitar sus propios diagramas (como ocurre en un sistema de control). También puede ser necesario describir las estructuras de datos, en sistemas o subsistemas en los que los datos sean un factor dominante (como una base de datos).

Sabemos que la Fase 2 ha acabado cuando se han descrito los objetos y sus interfaces. Bueno, la mayoría —hay generalmente unos pocos que quedan ocultos y que no se dan a conocer hasta la Fase 3. Pero esto es correcto. En lo que a uno respecta, esto es todo lo que se ha podido descubrir de los objetos a manipular. Es bonito descubrirlos en las primeras etapas del proceso pero la POO proporciona una estructura tal, que no presenta problema si se descubren más tarde. De hecho, el diseño de un objeto tiende a darse en cinco etapas, a través del proceso completo de desarrollo de un programa.

---

[18] Para los principiantes, recomiendo *UML Distilled*, 2.ª edición.

[19] Python *(http://www.Python.org)* suele utilizarse como "pseudocódigo ejecutable".

Las cinco etapas del diseño de un objeto

La duración del diseño de un objeto no se limita al tiempo empleado en la escritura del programa, sino que el diseño de un objeto conlleva una serie de etapas. Es útil tener esta perspectiva porque se deja de esperar la perfección; por el contrario, uno comprende lo que hace un objeto y el nombre que debería tener surge con el tiempo. Esta visión también se aplica al diseño de varios tipos de programas; el patrón para un tipo de programa particular emerge al enfrentarse una y otra vez con el problema (esto se encuentra descrito en el libro *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*). Los objetos, también tienen su patrón, que emerge a través de su entendimiento, uso y reutilización.

1. **Descubrimiento de los objetos**. Esta etapa ocurre durante el análisis inicial del programa. Se descubren los objetos al buscar factores externos y limitaciones, elementos duplicados en el sistema, y las unidades conceptuales más pequeñas. Algunos objetos son obvios si ya se tiene un conjunto de bibliotecas de clases. La comunidad entre clases que sugieren clases bases y herencia, puede aparecer también en este momento, o más tarde dentro del proceso de diseño.

2. **Ensamblaje de objetos**. Al construir un objeto se descubre la necesidad de nuevos miembros que no aparecieron durante el descubrimiento. Las necesidades internas del objeto pueden requerir de otras clases que lo soporten.

3. **Construcción del sistema**. De nuevo, pueden aparecer en esta etapa más tardía nuevos requisitos para el objeto. Así se aprende que los objetos van evolucionando. La necesidad de un objeto de comunicarse e interconectarse con otros del sistema puede hacer que las necesidades de las clases existentes cambien, e incluso hacer necesarias nuevas clases. Por ejemplo, se puede descubrir la necesidad de clases que faciliten o ayuden, como una lista enlazada, que contiene poca o ninguna información de estado y simplemente ayuda a la función de otras clases.

4. **Aplicación del sistema**. A medida que se añaden nuevos aspectos al sistema, puede que se descubra que el diseño previo no soporta una ampliación sencilla del sistema. Con esta nueva información, puede ser necesario reestructurar partes del sistema, generalmente añadiendo nuevas clases o nuevas jerarquías de clases.

5. **Reutilización de objetos**. Ésta es la verdadera prueba de diseño para una clase. Si alguien trata de reutilizarla en una situación completamente nueva, puede que descubra pequeños inconvenientes. Al cambiar una clase para adaptarla a más programas nuevos, los principios generales de la clase se mostrarán más claros, hasta tener un tipo verdaderamente reutilizable. Sin embargo, no debe esperarse que la mayoría de objetos en un sistema se diseñen para ser reutilizados —es perfectamente aceptable que un porcentaje alto de los objetos sean específicos del sistema para el que fueron diseñados. Los tipos reutilizables tienden a ser menos comunes, y deben resolver problemas más generales para ser reutilizables.

Guías para el desarrollo de objetos

Estas etapas sugieren algunas indicaciones que ayudarán a la hora de pensar en el desarrollo de clases:

1. Debe permitirse que un problema específico genere una clase, y después dejar que la clase crezca y madure durante la solución de otros problemas.

2. Debe recordarse que descubrir las clases (y sus interfaces) que uno necesita es la tarea principal del diseño del sistema. Si ya se disponía de esas clases, el proyecto será fácil.

3. No hay que forzar a nadie a saber todo desde el principio; se aprende sobre la marcha.Y esto ocurrirá poco a poco.

4. Hay que empezar programando; es bueno lograr algo que funcione de manera que se pueda probar la validez o no de un diseño. No hay que tener miedo a acabar con un código de estilo procedimental malo —las clases segmentan el problema y ayudan a controlar la anarquía y la entropía. Las clases malas no estropean las clases buenas.

5. Hay que mantener todo lo más simple posible. Los objetos pequeños y limpios con utilidad obvia son mucho mejores que interfaces grandes y complicadas. Cuando aparecen puntos de diseño puede seguirse el enfoque de una afeitadora de Occam: se consideran las alternativas y se selecciona la más simple, porque las clases simples casi siempre resultan mejor. Hay que empezar con algo pequeño y sencillo, siendo posible ampliar la interfaz de la clase al entenderla mejor. A medida que avance el tiempo será difícil eliminar elementos de una clase.

Fase 3: Construir el núcleo

Ésta es la conversión inicial de diseño pobre en un código compilable y ejecutable que pueda ser probado, y especialmente, que pueda probar la validez o no de la arquitectura diseñada. Este proceso no se puede hacer de una pasada, sino que consistirá más bien en una serie de pasos que permitirán construir el sistema de manera iterativa, como se verá en la Fase 4.

Su objetivo es encontrar el núcleo de la arquitectura del sistema que necesita implementar para generar un sistema ejecutable, sin que importe lo incompleto que pueda estar este sistema en esta fase inicial. Está creando un armazón sobre el que construir en posteriores iteraciones. También se está llevando a cabo la primera de las muchas integraciones y pruebas del sistema, a la vez que proporcionando a los usuarios una realimentación sobre la apariencia que tendrá su sistema, y cómo va progresando. Idealmente, se están además asumiendo algunos riesgos críticos. De hecho, se descubrirán posibles cambios y mejoras que se pueden hacer sobre el diseño original —cosas que no se hubieran descubierto de no haber implementado el sistema.

Una parte de la construcción del sistema es comprobar que realmente se cumple el análisis de requisitos y la especificación del sistema que realmente cumple el analisis de requisitos y la especificación del sistema (independientemente de la forma en que estén planteados). Debe asegurarse que las pruebas verifican los requerimientos y los casos de uso. Cuando el corazón del sistema sea estable, será posible pasar a la siguiente fase y añadir nuevas funcionalidades.

Fase 4: Iterar los casos de uso

Una vez que el núcleo del sistema está en ejecución, cada característica que se añada es en sí misma un pequeño proyecto. Durante cada *iteración*, entendida como un periodo de desarrollo razonablemente pequeño, se añade un conjunto de características.

¿Cúal debe ser la duración de una iteración? Idealmente, cada iteración dura de una a tres semanas (la duración puede variar en función del lenguaje de implementación). Al final de ese periodo, se tiene un sistema integrado y probado con una funcionalidad mayor a la que tenía previamente. Pero lo particularmente interesante es la base de la iteración: un único caso de uso. Cada caso de uso es un paquete de funcionalidad relacionada que se construye en el sistema de un golpe, durante una iteración. Esto no sólo proporciona una mejor idea de lo que debería ser el ámbito de un caso de uso, sino que además proporciona una validación mayor de la idea del mismo, dado que el concepto no queda descartado hasta después del análisis y del diseño, pues éste es una unidad de desarrollo fundamental a lo largo de todo el proceso de construcción de software.

Se deja de iterar al lograr la funcionalidad objetivo, o si llega un plazo y el cliente se encuentra satisfecho con la versión actual (debe recordarse que el software es un negocio de suscripción). Dado que el proceso es iterativo, uno puede tener muchas oportunidades de lanzar un producto, más que tener un único punto final; los proyectos abiertos trabajan exclusivamente en entornos iterativos de gran nivel de realimentación, que es precisamente lo que les permite acabar con éxito.

Un proceso de desarrollo iterativo tiene gran valor por muchas razones. Si uno puede averiguar y resolver pronto los riesgos críticos, los clientes pueden tener muchas oportunidades de cambiar de idea, la satisfacción del programador es mayor, y el proyecto puede guiarse con mayor precisión. Pero otro beneficio adicional importante es la realimentación a los usuarios, que pueden ver a través del estado actual del producto cómo va todo. Así es posible reducir o eliminar la necesidad de reuniones de estado "entumece-mentes" e incrementar la confianza y el soporte de los usuarios.

Fase 5: Evolución

Éste es el punto del ciclo de desarrollo que se ha denominado tradicionalmente "mantenimiento", un término global que quiere decir cualquier cosa, desde "hacer que funcione de la manera que se suponía que lo haría en primer lugar", hasta "añadir aspectos varios que el cliente olvidó mencionar", pasando por el tradicional "arreglar los errores que puedan aparecer" o "la adición de nuevas características a medida que aparecen nuevas necesidades". Por ello, al término "mantenimiento" se le han aplicado numerosos conceptos erróneos, lo que ha ocasionado un descenso progresivo de su calidad, en parte porque sugiere que se construyó una primera versión del programa en la cual hay que ir cambiando partes, además de engrasarlo para evitar que se oxide. Quizás haya un término mejor para describir lo que está pasando.

Prefiero el término *evolución*[20]. De esta forma, "uno no acierta a la primera, por lo que debe concederse la libertad de aprender y volver a hacer nuevos cambios". Podríamos necesitar muchos cambios a medida que vamos aprendiendo y comprendiendo con más detenimiento el problema. A corto y largo plazo, será el propio programa el que se verá beneficiado de este proceso continuo de evolución. De hecho, ésta permitirá que el programa pase de bueno a genial, haciendo que se aclaren aquellos aspectos que no fueron verdaderamente entendidos en la primera pasada. También es

---

[20] El libro de Martin Fowler *Refactoring: improving the design of existing code* (Addison-Wesley, 1999) cubre al menos un aspecto de la evolución, utilizando exclusivamente ejemplos en Java.

en este proceso en el que las clases se convierten en recursos reutilizables, en vez de clases diseñadas para su uso en un solo proyecto.

"Hacer el proyecto bien" no sólo implica que el programa funcione de acuerdo con los requisitos y casos de uso. También quiere decir que la estructura interna del código tenga sentido, y que parezca que encaja bien, sin aparentar tener una sintaxis extraña, objetos de tamaño excesivo o con fragmentos inútiles de código. Además, uno debe tener la sensación de que la estructura del programa sobrevivirá a los cambios que inevitablemente irá sufriendo a lo largo de su vida, y de que esos cambios se podrán hacer de forma sencilla y limpia. Esto no es trivial. Uno no sólo debe entender qué es lo que está construyendo, sino también cómo evolucionará el programa (lo que yo denomino el *vector del cambio*). Afortunadamente, los lenguajes de programación orientada a objetos son especialmente propicios para soportar este tipo de modificación continua —los límites creados por los objetos son los que tienden a lograr una estructura sólida. También permiten hacer cambios —que en un programa procedural parecerían drásticos— sin causar terremotos a lo largo del código. De hecho, el soporte a la evolución podría ser el beneficio más importante de la POO.

Con la evolución, se crea algo que al menos se aproxima a lo que se piensa que se está construyendo, se compara con los requisitos, y se ve dónde se ha quedado corto. Después, se puede volver y ajustarlo diseñando y volviendo a implementar las porciones del programa que no funcionaron correctamente[21]. De hecho, es posible que se necesite resolver un problema, o determinado aspecto de un problema, varias veces antes de dar con la solución correcta (suele ser bastante útil estudiar en este momento *el Diseño de Patrones)*. También es posible encontrar información en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*).

La evolución también se da al construir un sistema, ver que éste se corresponda con los requisitos, y descubrir después que no era, de hecho, lo que se pretendía. Al ver un sistema en funcionamiento, se puede descubrir que verdaderamente se pretendía que solucionase otro problema. Si uno espera que se dé este tipo de evolución, entonces se debe construir la primera versión lo más rápidamente posible con el propósito de averiguar sin lugar a dudas qué es exactamente lo que se desea.

Quizás lo más importante que se ha de recordar es que por defecto, si se modifica una clase, sus súper y subclases seguirán funcionando. Uno no debe tener miedo a la modificación (especialmente si se dispone de un conjunto de pruebas, o alguna prueba individual que permita verificar la corrección de las modificaciones). Los cambios no tienen por qué estropear el programa, sino que cualquiera de las consecuencias de un cambio se limitarán a las subclases y/o colaboradores específicos de la clase que se modifica.

Los planes merecen la pena

Por supuesto, uno jamás construiría una casa sin unos planos cuidadosamente elaborados. Si construyéramos un hangar o la casa de un perro, los planes no tendrían tanto nivel de detalle, pero pro-

[21] Esto es semejante a la elaboración de "prototipos rápidos", donde se supone que uno construye una versión "rápida y sucia" que permite comprender mejor el sistema, pero que es después desechada para construirlo correctamente. El problema con el prototipado rápido es que los equipos de desarrollo no suelen desechar completamente el prototipo, sino que lo utilizan como base sobre la que construir. Si se combina, en la programación procedural, con la falta de estructura, se generan sistemas totalmente complicados, y difíciles de mantener.

bablemente comenzaríamos con una serie de esbozos que nos permitiesen guiar el proceso. El desarrollo de software ha llegado a extremos. Durante mucho tiempo, la gente llevaba a cabo desarrollos sin mucha estructura, pero después, comenzaron a fallar los grandes procesos. Como reacción, todos acabamos con metodologías que conllevan una cantidad considerable de estructura y detalle, eso sí, diseñadas, en principio, para estos grandes proyectos. Estas metodologías eran demasiado tediosas de usar —parecía que uno invertiría todo su tiempo en escribir documentos, y que no le quedaría tiempo para programar (y esto ocurría a menudo). Espero haber mostrado aquí una serie de sugerencias intermedias. Independientemente de lo pequeño que sea, es necesario *algún* tipo de plan, que redundará en una gran mejora en el proyecto, especialmente respecto del que se obtendría si no se hiciera ningún plan de ningún tipo. Es necesario recordar que en muchas estimaciones, falla más del 50 por ciento del proyecto (¡incluso en ocasiones se llega al 70 por ciento!).

Siguiendo un plan —preferentemente uno simple y breve— y siguiendo una estructura de diseño antes de la codificación, se descubre que los elementos encajan mejor, de modo más sencillo que si uno se zambulle y empieza a escribir código sin ton ni son. También se alcanzará un nivel de satisfacción elevado. La experiencia dice que al lograr una solución elegante uno acaba completamente satisfecho, a un nivel totalmente diferente; uno se siente más cercano al arte que a la tecnología. Y la elegancia siempre merece la pena; no se trata de una persecución frívola. De hecho, no solamente proporciona un programa más fácil de construir y depurar, sino que éste es mucho más fácil de entender y mantener, que es precisamente donde reside su valor financiero.

Programación extrema

Una vez estudiadas las técnicas de análisis y diseño, por activa y por pasiva durante mucho tiempo, quizás el concepto de programación extrema (*Extreme Programming*, XP) sea el más radical y sorprendente que he visto. Es posible encontrar información sobre él mismo en *Extreme Programming Explained*, de Kent Beck (Addison-Wesley 2000), que puede encontrarse también en la Web en *http://www.xprogramming.com*.

XP es tanto una filosofía del trabajo de programación como un conjunto de guías para acometer esta tarea. Algunas de estas guías se reflejan en otras metodologías recientes, pero las dos contribuciones más distintivas e importantes en mi opinión son "escribir las pruebas en primer lugar" y "la programación a pares". Aunque Beck discute bastante todo el proceso en sí, señala que si se adoptan únicamente estas dos prácticas, uno mejorará enormemente su productividad y nivel de confianza.

Escritura de las pruebas en primer lugar

El proceso de prueba casi siempre ha quedado relegado al final de un proyecto, una vez que "se tiene todo trabajando, pero hay que asegurarlo". Implícitamente, tenía una prioridad bastante baja, y la gente que se especializa en las pruebas nunca ha gozado de un gran estatus, e incluso suele estar ubicada en el sótano, lejos de los "programadores de verdad". Los equipos de pruebas se han amoldado tanto a esta consideración que incluso han llegado a vestir de negro, y han chismorreado alegremente cada vez que lograban encontrar algún fallo (para ser honestos, ésta es la misma sensación que yo tenía cada vez que lograba encontrar algún fallo en un compilador).

XP revoluciona completamente el concepto de prueba dándole una prioridad igual (o incluso mayor) que a la codificación. De hecho, se escriben los tests *antes* de escribir el código a probar, y los códigos se mantienen para siempre junto con su código destino. Es necesario ejecutar con éxito los tests cada vez que se lleva a cabo un proceso de integración del proyecto (lo cual ocurre a menudo, en ocasiones más de una vez al día).

Al principio la escritura de las pruebas tiene dos efectos extremadamente importantes.

El primero es que fuerza una definición clara de la interfaz de cada clase. Yo, en numerosas ocasiones he sugerido que la gente "imagine la clase perfecta para resolver un problema particular" como una herramienta a utilizar a la hora de intentar diseñar el sistema. La estrategia de pruebas XP va más allá —especifica exactamente qué apariencia debe tener la clase para el consumidor de la clase, y cómo ésta debe comportarse exactamente. No puede haber nada sin concretar. Es posible escribir toda la prosa o crear todos los diagramas que se desee, describiendo cómo debería comportarse una clase, pero nada es igual que un conjunto de pruebas. Lo primero es una lista de deseos, pero las pruebas son un contrato reforzado por el compilador y el programa en ejecución. Cuesta imaginar una descripción más exacta de una clase que la de los tests.

Al crear los tests, uno se ve forzado a pensar completamente en la clase, y a menudo, descubre la funcionalidad deseada que podría haber quedado en el tintero durante las experiencias de pensamiento de los diagramas XML, las tarjetas CRC, los casos de uso, etc.

El segundo efecto importante de escribir las pruebas en primer lugar, proviene de la ejecución de las pruebas cada vez que se construye un producto software. Esta actividad proporciona la otra mitad de las pruebas que lleva a cabo el compilador. Si se observa la evolución de los lenguajes de programación desde esta perspectiva, se llegará a la conclusión de que las verdaderas mejoras en lo que a tecnología se refiere han tenido que ver con las pruebas. El lenguaje ensamblador solamente comprobaba la sintaxis, pero C imponía algunas restricciones semánticas, que han evitado que se produzca cierto tipo de errores. Los lenguajes POO imponen incluso más restricciones semánticas, que miradas así no son, de hecho, sino métodos de prueba. "¿Se está utilizando correctamente este tipo de datos?", y "¿se está invocando correctamente a esta función?" son algunos de los tipos de preguntas que hace un compilador o un sistema en tiempo de ejecución. Se han visto los resultados de tener estas pruebas ya incluidas en el lenguaje: la gente parece ser capaz de escribir sistemas más completos y hacer que funcionen, con menos cantidad de tiempo y esfuerzo. He intentado siempre averiguar la razón, pero ahora lo tengo claro, son las pruebas: cada vez que se hace algo mal, la red de pruebas de seguridad integradas dice que hay un problema y determina dónde.

Pero las pruebas integradas permitidas por el diseño del lenguaje no pueden ir mucho más allá. En cierto punto, *cada uno* debe continuar y añadir el resto de pruebas que producen una batería de pruebas completa (en cooperación con el compilador y el sistema en tiempo de ejecución) que verifique todo el programa. Y, exactamente igual que si se dispusiera de un compilador observando por encima del hombro, ¿no desearía uno que estas pruebas le ayudasen a hacer todo bien desde el principio? Por eso es necesario escribir las pruebas en primer lugar y ejecutarlas cada vez que se reconstruya el sistema. Las pruebas se convierten en una extensión de la red de seguridad proporcionada por el lenguaje.

Una de las cosas que he descubierto respecto del uso de lenguajes de programación cada vez más y más potentes es que conducen a la realización de experimentos cada vez más duros, pues se sabe a priori que el propio lenguaje evitará pérdidas innecesarias de tiempo en la localización de errores. El esquema de pruebas XP hace lo mismo para todo el proyecto. Dado que se sabe que las pruebas localizarán cualquier problema que pueda aparecer en la vida del proyecto (y cada vez que se nos ocurra alguno), simplemente se introducen nuevas pruebas), es posible hacer cambios, incluso grandes, cuando sea necesario sin preocuparse de que éstos puedan cargarse todo el proyecto. Esto es increíblemente potente.

Programación a pares

La programación a pares (por parejas) va más allá del férreo individualismo al que hemos sido adoctrinados desde el principio, a través de las escuelas (donde es uno mismo el que fracasa o tiene éxito), de los medios de comunicación, especialmente las películas de Hollywood, en las que el héroe siempre lucha contra la conformidad sin sentido[22]. Los programadores, también, suelen considerarse abanderados de la individualidad —"los vaqueros codificadores" como suele llamarlos Larry Constantine. Y por el contrario, XP, que trata, de por sí, de luchar contra el pensamiento convencional, enuncia lo contrario, afirmando que el código debería siempre escribirse entre dos personas por cada estación de trabajo. Y esto debería hacerse en áreas en las que haya grupos de estaciones de trabajo, sin las barreras de las que la gente de facilidades de diseño suelen estar tan orgullosos. De hecho, Beck dice que la primera tarea para convertirse a XP es aparecer con destornilladores y llaves Allen y desmontar todo aquello que parezca imponer barreras o separaciones[23] (esto exige contar con un director capaz de hacer frente a todas las quejas del departamento de infraestructuras).

El valor de la programación en pareja es que una persona puede estar, de hecho, codificando mientras la otra piensa en lo que se está haciendo. El pensador es el que tiene en la cabeza todo el esbozo —y no sólo una imagen del problema que se está tratando en ese momento, sino todas las guías del XP. Si son dos las personas que están trabajando, es menos probable que uno de ellos huya diciendo "No quiero escribir las pruebas lo primero", por ejemplo. Y si el codificador se queda clavado, pueden cambiar de sitio. Si los dos se quedan parados, puede que alguien más del área de trabajo pueda contribuir al oír sus meditaciones. Trabajar a pares hace que todo fluya mejor y a tiempo. Y lo que probablemente es más importante: convierte la programación en una tarea mucho más divertida y social.

He comenzado a hacer uso de la programación en pareja durante los periodos de ejercitación en algunos de mis seminarios, llegando a la conclusión de que mejora significativamente la experiencia de todos.

---

[22] Aunque probablemente ésta sea más una perspectiva americana, las historias de Hollywood llegan a todas partes.

[23] Incluido (especialmente) el sistema PA. Trabajé una vez en una compañía que insistía en difundir a todo el mundo cualquier llamada entrante que recibieran los ejecutivos, lo cual interrumpía continuamente la productividad del equipo (pero los directores no podían empezar siquiera a pensar en prescindir de un servicio tan importante como el PA). Al final, y cuando nadie me veía, me encargué de cortar los cables de los altavoces.

Por qué Java tiene éxito

La razón por la que Java ha tenido tanto éxito es que su propósito era resolver muchos de los problemas a los que los desarrolladores se enfrentan hoy en día. El objetivo de Java es mejorar la productividad. Esta productividad se traduce en varios aspectos, pero el lenguaje fue diseñado para ayudar lo máximo posible, dejando en manos de cada uno la mínima cantidad posible, tanto de reglas arbitrarias, como de requisitos a usar en determinados conjuntos de aspectos. Java fue diseñado para ser práctico; las decisiones de diseño del lenguaje Java se basaban en proporcionar al programador la mayor cantidad de beneficios posibles.

Los sistemas son más fáciles de expresar y entender

Las clases diseñadas para encajar en el problema tienden a expresarlo mejor. Esto significa que al escribir el código uno está describiendo su solución en términos del espacio del problema, en vez de en términos del computador, que es el espacio de la solución ("Pon el bit en el chip que indica que el relé se va cerrar"). Uno maneja conceptos de alto nivel y puede hacer mucho más con una única línea de código.

El otro beneficio del uso de esta expresión es la mantenibilidad que (si pueden creerse los informes) se lleva una porción enorme del coste de un programa durante toda su vida. Si un programa es fácil de entender, entonces es fácil de mantener. Esto también puede reducir el coste de crear y mantener la documentación.

Ventajas máximas con las bibliotecas

La manera más rápida de crear un programa es utilizar código que ya esté escrito: una biblioteca. Uno de los principales objetivos de Java es facilitar el uso de bibliotecas. Esta meta se logra convirtiendo las bibliotecas en nuevos tipos de datos (clases), de forma que la incorporación de una biblioteca equivale a la inserción de nuevos tipos al lenguaje. Dado que el compilador de Java se encarga del buen uso de las bibliotecas —garantizando una inicialización y eliminación completas, y asegurando que se invoca correctamente a las funciones— uno puede centrarse en lo que desea que haga la biblioteca en vez de cómo tiene que hacerlo.

Manejo de errores

El manejo de errores en C es un importante problema, que suele ser frecuentemente ignorado o que se trata de evitar cruzando los dedos. Si se está construyendo un programa grande y complejo, no hay nada peor que tener un error enterrado en algún sitio sin tener ni siquiera una pista de dónde puede estar. El *manejo de excepciones* de Java es una forma de garantizar que se notifiquen los errores, y que todo ocurre como consecuencia de algo.

Programación a lo grande

Muchos lenguajes de programación "tradicionales" tenían limitaciones intrínsecas en lo que al tamaño y complejidad del programa se refiere. BASIC, por ejemplo, puede ser muy bueno para poner juntas soluciones rápidas para cierto tipo de problemas, pero si el programa se hace mayor de varias páginas, o se sale del dominio normal del problema, es como intentar nadar en un fluido cada vez más viscoso. No hay una línea clara que permita separar cuándo está fallando el lenguaje, y si la hubiera, la ignoraríamos. Uno no dice "Mi programa en BASIC simplemente creció demasiado; tendré que volver a escribirlo en C". Más bien se intenta meter con calzador unas pocas líneas para añadir alguna nueva característica. Por tanto, el coste extra viene dependiendo de uno mismo.

Java está diseñado para ayudar a *programar a lo grande* —es decir, para borrar esos límites de complejidad entre un programa pequeño y uno grande. Uno no tiene por qué usar POO al escribir un programa de utilidad del estilo de "¡Hola, mundo!", pero estas características siempre están ahí cuando son necesarias. Y el compilador se muestra agresivo a la hora de descubrir las causas generadoras de errores, tanto en el caso de programas grandes, como pequeños.

Estrategias para la transición

Si uno se introduce en la POO, la siguiente pregunta será probablemente "¿Cómo puedo hacer que mi director, mis colegas, mi departamento,... empiecen a usar objetos?". Uno debe pensar en cómo él mismo —un programador independiente— se sentiría a la hora de aprender un nuevo lenguaje y un nuevo paradigma de programación. A fin de cuentas, ya lo ha hecho antes. Lo primero es la educación y el uso de ejemplos, después viene un proyecto de prueba que proporcione una idea clara de los fundamentos sin hacer algo demasiado confuso. Después viene un proyecto "del mundo real" que, de hecho, haga algo útil. A lo largo de los primeros proyectos, uno sigue su educación leyendo y preguntando a los expertos, a la vez que solucionando pequeños inconvenientes con los colegas. Éste es el enfoque que muchos programadores experimentados sugieren de cara a migrar a Java. Cambiar una compañía entera, por supuesto implicaría la introducción de alguna dinámica de grupo, pero ayudará a recordar en cada paso cómo debería desenvolverse cada uno.

Guías

He aquí algunas ideas o guías a tener en cuenta cuando se haga la transición a POO y Java:

1. Formación

El primer paso es algún tipo de educación. Hay que recordar la inversión en código de la compañía, e intentar no tirar todo a la basura durante los seis a nueve meses que lleve a todo el mundo enterarse de cómo funcionan las interfaces. Es mejor seleccionar un pequeño grupo para adoctrinarles, compuesto preferentemente por personas curiosas, y que trabajen bien en grupo, que pueda luego funcionar como una red de soporte propia mientras se esté aprendiendo Java.

Un enfoque alternativo recomendado en ocasiones, es formar a todos los niveles de la compañía a la vez, incluidos cursos muy por encima para los directores de estrategia, además de cursos de diseño y programación para los constructores de proyectos. Esto es especialmente bueno para las pequeñas compañías que cambian continuamente la manera de hacer las cosas, o a nivel de divisiones en aquellas compañías de gran tamaño. Dado que el coste es elevado, sin embargo, hay que elegir empezar de alguna manera con la formación a nivel de proyecto, llevar a cabo un proyecto piloto (posiblemente con un formador externo) y dejar que el equipo de proyecto se convierta en el grupo de profesores del resto de la compañía.

2. Proyecto de bajo riesgo

Es necesario empezar con un proyecto de bajo riesgo y permitir los errores. Una vez que se ha adquirido cierta experiencia, uno puede alimentarse bien de proyectos de miembros del mismo equipo, o bien utilizar a los miembros del equipo como personal de soporte técnico para POO. Puede que el primer proyecto no funcione correctamente a la primera, por lo que no debería ser crítico con la misión de la compañía. Debería ser simple, independiente, e instructivo; esto significa que debería conllevar la creación de clases con significado para cuando les llegue el turno de aprender Java al resto de empleados de la compañía.

3. Modelo que ya ha tenido éxito

Es necesario buscar ejemplos con un buen diseño orientado a objetos en vez de empezar de la nada. Hay muchas posibilidades de que exista alguien que ya haya solucionado el problema en cuestión, o que si no lo ha solucionado del todo pueda aplicar lo ya aprendido sobre la abstracción para modificar un diseño ya existente en aras de que se ajuste a tus necesidades. Éste es el concepto general de los *patrones de diseño*, cubiertos en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com.*

4. Utilizar bibliotecas de clases existentes

La motivación económica principal para cambiar a POO es la facilidad de usar código ya existente en forma de bibliotecas de clases (en particular las bibliotecas estándares de Java, cubiertas completamente a lo largo de este libro). Se obtendrá el ciclo de desarrollo más pequeño posible cuando se puedan crear y utilizar objetos de bibliotecas preconfeccionadas. Sin embargo, algunos programadores novatos no entienden este concepto, y son inconscientes de la existencia de bibliotecas de clases. El éxito con la POO y Java será óptimo si se hace un esfuerzo para buscar y reutilizar el código ya desarrollado cuanto antes en el proceso de transición.

5. No reescribir en Java código ya existente

No suele ser la mejor de las ideas tomar código ya existente y que funcione y reescribirlo en Java (si se convierten en objetos, es posible interactuar con código ya escrito en C o C++ haciendo uso de la Interfaz Nativa Java (*Java Native Interface*) descrito en el Apéndice B). Hay beneficios incrementales, especialmente si el código va a ser reutilizado. Pero todas las opciones pasan porque no se darán los incrementos drásticos de productividad que uno pudiera esperar para su primer pro-

yecto a no ser que se acometa uno totalmente nuevo. Java y la POO brillan mucho más cuando se pasa de un proyecto conceptual al real correspondiente.

Obstáculos de gestión

Para aquél que sea director, su trabajo consiste en adquirir los recursos para el equipo, superar las barreras que puedan dificultar el éxito del equipo, y en general, intentar proporcionar el entorno más productivo que permita al equipo disfrutar y conseguir así, llevar a cabo esos milagros que siempre se exigen. Pasarse a Java implica estas tres características, y sería maravilloso que el coste fuera, además nulo. Aunque pasarse a Java puede ser más barato —en función de las limitaciones de cada uno— que las alternativas de la POO para un equipo de programadores en C (y probablemente para los programadores de otros lenguajes procedurales) no es gratuito, y hay obstáculos de los que uno debería ser consciente a la hora de vender el pasarse a Java dentro de una compañía y quedar totalmente embarrancado.

Costes iniciales

El coste de pasarse a Java es mayor que el de adquirir compiladores de Java (el compilador de Java de Sun es gratuito, así que éste difícilmente podría constituir un obstáculo). Los costes a medio y largo plazo se minimizan si se invierte en formación (y posiblemente si se utiliza un formador durante el primer proyecto) y también si se identifica y adquiere una biblioteca de clases que solucione el problema en vez de intentar construir esas bibliotecas uno mismo. Éstos son costes muy elevados que deben ser cuantificados en una propuesta realista. Además, están los costes ocultos de la pérdida de productividad implícita en el aprendizaje de un nuevo lenguaje, y probablemente un nuevo entorno de programación. La formación y la búsqueda de un formador pueden, a ciencia cierta, minimizar estos costes, pero los miembros del equipo deberán sobreponerse a sus propios problemas para comprender la nueva tecnología. Durante este proceso, ellos cometerán más fallos (éste es un aspecto importante, pues los errores reconocidos son la forma más rápida de aprender) y serán menos productivos. Incluso entonces, con algunos tipos de problemas de programación, las clases correctas y el entorno de desarrollo correcto, es posible ser más productivo mientras se está aprendiendo Java (incluso considerando que se están cometiendo más fallos y escribiendo menos líneas de código cada día) que si se continuara con C.

Aspectos de rendimiento

Una pregunta frecuente es "¿La POO hace que los programas se conviertan en más grandes y lentos automáticamente?". La respuesta es: "Depende". Los aspectos extra de seguridad de Java tradicionalmente han conllevado una penalización en el rendimiento, frente a lenguajes como C++. Las tecnologías como "hotspot" y las tecnologías de compilación han mejorado significativamente la velocidad en la mayoría de los casos, y se continúan haciendo esfuerzos para lograr un rendimiento aún mayor.

Cuando uno se centra en un prototipado rápido, es posible desechar conjuntamente componentes lo más rápido posible, a la vez que se ignoran ciertos aspectos de eficiencia. Si se utilizan bibliotecas de un tercero, éstas suelen estar optimizadas por el propio fabricante; en cualquier caso, esto no es un problema cuando uno está en modo de desarrollo rápido. Cuando se tiene el sistema deseado,

que sea lo suficientemente pequeño y rápido, entonces, ya está. Si no, hay que empezar a reescribir pequeñas porciones de código. Si a pesar de esto no se mejora, hay que pensar cómo hacer modificaciones en la implementacion subyacente, de forma que no haya ningún código que use una clase concreta que vaya a ser modificada. Sólo si no se encuentra ninguna otra solución al problema se acometerán posibles cambios en el diseño. El hecho de que el rendimiento sea crítico en esa porción del diseño es un indicador que debe formar parte del criterio de diseño principal. La utilización del desarrollo rápido ofrece la ventaja de poder averiguar esto muy pronto.

Si se encuentra una función que constituya un cuello de botella, es posible reescribirla en C/C++ haciendo uso de los *métodos nativos* de Java, sobre los que versa el Apéndice B.

Errores de diseño comunes

Cuando un equipo empieza a trabajar en POO y Java, los programadores cometerán una serie de errores de diseño comunes. Esto ocurre a menudo debido a que hay una realimentación insuficiente por parte de los expertos durante el diseño e implementación de los primeros proyectos, puesto que no han aparecido expertos dentro de la compañía y porque puede que haya cierta resistencia en la empresa para retener a los consultores. Es fácil que si alguien cree entender la POO desde las primeras etapas del ciclo trate de atajar a través de una tangente errónea. Algo que es obvio a los ojos de una persona experta en el lenguaje, puede llegar a constituir un gran problema o debate interno para un novato. Podría evitarse un porcentaje elevado de este trauma si se utilizara un experto externo experimentado como formador y consejero.

¿Java frente a C++?

Java se parece bastante a C++, y naturalmente podría parecer que C++ está siendo reemplazado por Java. Pero me empiezo a cuestionar esta lógica. Para algunas cosas, C++ sigue teniendo una serie de características que Java no tiene, y aunque ha habido muchas promesas de que algún día Java llegará a ser tan o más rápido que C++, hasta la fecha solamente hemos sido testigos de ligeras mejoras, sin innovaciones drásticas. También parece que sigue habiendo un interés continuo en C++, por lo que es improbable que este lenguaje desaparezca con el tiempo. (Los lenguajes siempre merodean por ahí. En uno de los "Seminarios de Java Intermedio/Avanzado" del autor Allen Holub afirmó que los lenguajes más comúnmente utilizados son Rexx y COBOL, en ese orden.)

Comienzo a pensar que la fuerza de Java reside en un ruedo ligeramente diferente al de C++. Éste es un lenguaje que no trata de encajar en un molde. Verdaderamente, se ha adaptado de distintas maneras para resolver problemas particulares. Algunas herramientas de C++ combinan bibliotecas, modelos de componente, y herramientas de generación de código para resolver el problema de desarrollar aplicaciones de ventanas para usuarios finales (para Microsoft Windows). Y sin embargo, ¿qué es lo que utilizan la gran mayoría de desarrolladores en Windows? Visual Basic (VB) de Microsoft. Y esto a pesar del hecho de que VB produce el tipo de código que se convierte en inmanejable en cuanto el programa tiene una ampliación de unas pocas páginas (además de proporcionar una sintaxis que puede ser incluso mística). VB es tan mal ejemplo de lenguaje de diseño como exitoso y popular. Por ello sería bueno disponer de la facilidad y potencia de VB sin que el resultado fuera código imposible de gestionar. Y es aquí donde Java debería destacar: como el "próxi-

mo VB". Uno puede estremecerse al leer esto, o no, pero al menos debería pensar en ello: es tan grande la porción de Java diseñada para facilitar la tarea del programador a la hora de enfrenarse a problemas de nivel de aplicación como las redes o las interfaces de usuario multiplataforma, y además tiene un diseño de lenguaje que hace posible la creación de bloques de código flexibles y de gran tamaño. Si se añade a esto el hecho de que Java es el lenguaje con los sistemas de comprobación de tipos y manejo de errores más robustos jamás vistas en un lenguaje, se tienen las bases para dar un gran paso adelante en lo que se refiere a productividad de la programación.

¿Debería utilizarse Java en vez de C++ para un proyecto determinado? En vez de *applets* de web, deben considerarse dos aspectos. El primero es que si se desea utilizar muchas de las bibliotecas de C++ ya existentes (logrando una considerable ganancia en productividad) o si se dispone de un código base ya existente en C o C++, Java podría ralentizar el desarrollo en vez de acelerarlo.

Si se está desarrollando el código por primera vez desde la nada, la simplicidad de Java frente a C++ acortará significativamente el tiempo de desarrollo —la evidencia anecdótica (historias de equipos que desarrollan en C++ y que siempre cuento a aquéllos que se pasan a Java) sugiere que se doble la velocidad de desarrollo frente a C++. Si el rendimiento de Java no importa o puede compensarse, los aspectos puramente de tiempo de lanzamiento hacen difícil justificar la elección de C++ frente a Java.

El aspecto más importante es el rendimiento. El código Java interpretado siempre ha sido lento, incluso entre 20 y 50 veces más lento que C en el caso de los primeros intérpretes de Java. Este aspecto, no obstante, ha mejorado considerablemente a lo largo del tiempo, aunque sigue siendo del orden de varias veces superior. Los computadores se fundamentan en la velocidad; si hacer algo en un computador no es considerablemente más rápido, lo hacemos a mano. (Incluso se sugiere que se empiece con Java, para reducir el tiempo de desarrollo, para posteriormente utilizar una herramienta y bibliotecas de soporte que permitan traducir el código a C++, cuando se necesite una velocidad de ejecución más rápida.)

La clave para hacer Java adecuado para la mayoría de proyectos de desarrollo es la aparición de mejoras en cuanto a velocidad, como los denominados compiladores *just-in-time* (JIT), la tecnología "hotspot" de Sun, e incluso compiladores de código nativo. Por supuesto, estos últimos eliminan la ejecución multiplataforma de los programas compilados, pero también proporcionan una mejora de velocidad al ejecutable, que se acerca a la que se lograría con C y C++. Y compilar un programa multiplataforma en Java sería bastante más sencillo que hacerlo en C o C++. (En teoría, simplemente es necesario recompilar, pero esto ya se ha prometido también antes en otros lenguajes de programación).

Es posible encontrar comparaciones entre Java y C++, y observaciones sobre las realidades de Java en los apéndices de la primera edición de este libro (disponible en el CD ROM que acompaña al presente texto, además de en *http://www.BruceEckel.com)*.

Resumen

Este capítulo trata de dar un repaso a los aspectos más importantes de la programación orientada a objetos y Java, incluyendo el porqué la POO es diferente, y por qué Java en particular es diferente,

conceptos de metodologías de POO, y finalmente las situaciones que se dan al hacer que una compañía pase a POO y Java.

La POO y Java pueden no ser para todo el mundo. Es importante evaluar las propias necesidades y decidir si Java podría satisfacer completamente esas necesidades, o si no sería mejor hacer uso de otro sistema de programación (incluyendo el que se esté utilizando actualmente). Si se sabe que las necesidades serán muy especializadas en un futuro próximo y que se tienen limitaciones específicas, puede que Java no sea la solución más satisfactoria, por lo que uno debe investigar las posibles alternativas[24]. Incluso si eventualmente se elige Java como lenguaje, uno debe al menos entender cuáles eran las opciones y tener una visión clara de por qué eligió dirigirse en esa dirección.

La apariencia de un lenguaje de programación procedural es conocida: definiciones de datos y llamadas a funciones. Para averiguar el significado de estos programas hay que invertir cierto tiempo, echando un vistazo a las llamadas a función y a conceptos de bajo nivel para crearse un modelo en la mente. Ésta es la razón por la que son necesarias representaciones intermedias al diseñar programas procedurales —por sí mismos, estos programas tienden a ser confusos porque los términos de expresión suelen estar más orientados hacia el computador que hacia el problema que se trata de resolver.

Dado que Java añade muchos conceptos nuevos sobre lo que tienen los lenguajes procedurales, es algo natural pensar que el método **main( )** de un programa en Java será bastante más complicado que su equivalente en un programa en C. Se verá que las definiciones de los objetos que representan conceptos en el espacio del problema (en vez de hacer uso de aspectos de representación del computador) además de los mensajes que se envían a los mismos, representan las actividades en ese mismo espacio. Una de las maravillas de la programación orientada a objetos es ésa: con un programa bien diseñado, es fácil entender el código simplemente leyéndolo. Generalmente hay también menos código, porque muchos de los problemas se resolverán reutilizando código de las bibliotecas ya existentes.

[24] Recomiendo, en particular, echar un vistazo a Python (http:(//www.Python.org).

2: Todo es un objeto

Aunque se basa en C++, Java es más un lenguaje orientado a objetos "puro".

Tanto C++ como Java son lenguajes híbridos, pero en Java los diseñadores pensaban que esa "hibridación" no era tan importante como lo era en C++. Un lenguaje híbrido permite múltiples estilos de programación; la razón por la que C++ es híbrido es soportar la compatibilidad hacia atrás con el lenguaje C. Dado que C++ es un superconjunto del lenguaje C, incluye muchas de las características no deseables de ese lenguaje, lo que puede provocar que algunos aspectos de C++ sean demasiado complicados.

El lenguaje Java asume que se desea llevar a cabo exclusivamente programación orientada a objetos. Esto significa que antes de empezar es necesario cambiar la forma de pensar hacia el mundo de la orientación a objetos (a menos que ya esté en él). El beneficio de este esfuerzo inicial es la habilidad para programar en un lenguaje que es más fácil de aprender y usar que otros muchos lenguajes de POO. En este capítulo, veremos los componentes básicos de un programa Java y aprenderemos que todo en Java es un objeto, incluido un programa Java.

Los objetos se manipulan mediante referencias

Cada lenguaje de programación tiene sus propios medios de manipular datos. Algunas veces, el programador debe ser consciente constantemente del tipo de manipulación que se está produciendo. ¿Se está manipulando directamente un objeto, o se está tratando con algún tipo de representación indirecta (un puntero en C o C++) que debe ser tratada con alguna sintaxis especial?

Todo esto se simplifica en Java. Todo se trata como un objeto, de forma que hay una única sintaxis consistente que se utiliza en todas partes. Aunque se *trata* todo como un objeto, el identificador que se manipula es una "referencia" a un objeto[1]. Se podría imaginar esta escena como si se tratara de

[1] Esto puede suponer un tema de debate. Existe quien piensa que "claramente, es un puntero", pero esto presupone una implementación subyacente. Además, las referencias de Java son mucho más parecidas en su sintaxis a las referencias de C++ que a punteros. En la primera edición del presente libro, el autor decidió inventar un nuevo término, "empuñadura" porque las referencias C++ y las referencias Java tienen algunas diferencias importantes. El autor provenía de C++ y no deseaba confundir a los programadores de C++ que supuestamente serían la mejor audiencia para Java. En la 2ª edición, el autor decidió que el término más comúnmente usado era el término "referencia", y que cualquiera que proviniera de C++ tendría que lidiar con mucho más que con la terminología de las referencias, por lo que podrán incorporarse sin problemas. Sin embargo, hay personas que no están de acuerdo siquiera con el término "referencia". El autor leyó una vez un libro en el que "era incorrecto decir que Java soporta el paso por referencia", puesto que los identificadores de objetos en Java (en concordancia con el citado autor) son *de hecho* "referencias a objetos". "Y (continúa el citado texto), todo se pasa *de hecho* por valor. Por tanto, si no se pasan parámetros por referencia, se está pasando una referencia a un objeto por valor". Se podría discutir la precisión de semejantes explicaciones, pero el autor considera que su enfoque simplifica el entendimiento del concepto sin herir a nadie (bueno, los abogados del lenguaje podrían decir que el autor miente, pero creo que la abstracción que se presenta es bastante apropiada).

una televisión (el objeto) con su mando a distancia (la referencia). A medida que se hace uso de la referencia, se está conectado a la televisión, pero cuando alguien dice "cambia de canal" o "baja el volumen", lo que se manipula es la referencia, que será la que manipule el objeto. Si desea moverse por la habitación y seguir controlando la televisión, se toma el mando a distancia (la referencia), en vez de la televisión.

Además, el mando a distancia puede existir por sí mismo, aunque no haya televisión. Es decir, el mero hecho de tener una referencia no implica necesariamente la existencia de un objeto conectado al mismo. De esta forma si se desea tener una palabra o frase, se crea una referencia **String**:

```
String s;
```

Pero esta sentencia *solamente* crea la referencia, y no el objeto. Si se decide enviar un mensaje a **s** en este momento, se obtendrá un error (en tiempo de ejecución) porque **s** no se encuentra, de hecho, vinculado a nada (no hay televisión). Una práctica más segura, por consiguiente, es inicializar la referencia en el mismo momento de su creación:

```
String s = "asdf";
```

Sin embargo, esta sentencia hace uso de una característica especial de Java: las cadenas de texto pueden inicializarse con texto entre comillas. Normalmente, es necesario usar un tipo de inicialización más general para los objetos.

Uno debe crear todos los objetos

Cuando se crea una referencia, se desea conectarla con un nuevo objeto. Así se hace, en general, con la palabra clave **new**, que dice "Créame un objeto nuevo de ésos". Por ello, en el ejemplo anterior se puede decir:

```
String s = new String ("asdf");
```

Esto no sólo significa "Créame un nuevo **String**", sino que también proporciona información sobre *cómo* crear el **String** proporcionando una cadena de caracteres inicial.

Por supuesto, **String** no es el único tipo que existe. Java viene con una plétora de tipos predefinidos. Lo más importante es que uno puede crear sus propios tipos. De hecho, ésa es la actividad fundamental de la programación en Java, y es precisamente lo que se irá aprendiendo en este libro.

Dónde reside el almacenamiento

Es útil visualizar algunos aspectos relativos a cómo se van disponiendo los elementos al ejecutar el programa, y en particular, sobre cómo se dispone la memoria. Hay seis lugares diferentes en los que almacenar información:

1. **Registros.** Son el elemento de almacenamiento más rápido porque existen en un lugar distinto al de cualquier otro almacenamiento: dentro del procesador. Sin embargo, el número de registros está severamente limitado, de forma que los registros los va asignando el compilador en función de sus necesidades. No se tiene control directo sobre ellos, y tampoco hay ninguna evidencia en los programas de que los registros siquiera existan.

2. **La pila.** Reside en la memoria RAM (memoria de acceso directo) general, pero tiene soporte directo del procesador a través del *puntero de pila*. Éste se mueve hacia abajo para crear más memoria y de nuevo hacia arriba para liberarla. Ésta es una manera extremadamente rápida y eficiente de asignar espacio de almacenamiento, antecedido sólo por los registros. El compilador de Java debe saber, mientras está creando el programa, el tamaño exacto y la vida de todos los datos almacenados en la pila, pues debe generar el código necesario para mover el puntero hacia arriba y hacia abajo. Esta limitación pone límites a la flexibilidad de nuestros programas, de forma que mientras existe algún espacio de almacenamiento en la pila —referencias a objetos en particular— los propios objetos Java no serán ubicados en la pila.

3. **El montículo.** Se trata de un espacio de memoria de propósito general (ubicado también en el área RAM) en el que residen los objetos Java. Lo mejor del montículo es que, a diferencia de la pila, el compilador no necesita conocer cuánto espacio de almacenamiento necesita asignar al montículo o durante cuánto tiempo debe permanecer ese espacio dentro del montículo. Por consiguiente, manejar este espacio de almacenamiento proporciona una gran flexibilidad. Cada vez que se desee crear un objeto, simplemente se escribe el código, se crea utilizando la palabra **new**, y se asigna el espacio de almacenamiento en el montículo en el momento en que este código se ejecuta. Por supuesto hay que pagar un precio a cambio de esta flexibilidad: lleva más tiempo asignar espacio de almacenamiento del montículo que lo que lleva hacerlo en la pila (es decir, si se *pudieran* crear objetos en la pila en Java, como se hace en C++).

4. **Almacenamiento estático.** El término "estático" se utiliza aquí con el sentido de "con una ubicación/posición fija" (aunque también sea en RAM). El almacenamiento estático contiene datos que están disponibles durante todo el tiempo que se esté ejecutando un programa. Podemos usar la palabra clave **static** para especificar que un elemento particular de un objeto sea estático, pero los objetos en sí nunca se sitúan en el espacio de almacenamiento estático.

5. **Almacenamiento constante.** Los valores constantes se suelen ubicar directamente en el código del programa, que es seguro, dado que estos valores no pueden cambiar. En ocasiones, las constantes suelen ser *acordonadas* por sí mismas, de forma que puedan ser opcionalmente ubicadas en memoria de sólo lectura (ROM).

6. **Almacenamiento no-RAM.** Si los datos residen completamente fuera de un programa, pueden existir mientras el programa no se esté ejecutando, fuera del control de dicho programa. Los dos ejemplos principales de esto son los objetos de flujo de datos *(stream)*, que se convierten en flujos o corrientes de bytes, generalmente para ser enviados a otra máquina, y los *objetos persistentes*, que son ubicados en el disco para que mantengan su estado incluso cuando el programa ha terminado. El truco con estos tipos de almacenamiento es convertir los objetos en algo que pueda existir en otro medio, y que pueda así recuperarse en forma de objeto basado en RAM cuando sea necesario. Java proporciona soporte para *persistencia ligera*, y

las versiones futuras de Java podrían proporcionar soluciones aún más complejas para la persistencia.

Un caso especial: los tipos primitivos

Hay un grupo de tipos que tiene un tratamiento especial: se trata de los tipos "primitivos", que se usarán frecuentemente en los programas. La razón para el tratamiento especial es que crear un objeto con **new** —especialmente variables pequeñas y simples— no es eficiente porque **new** coloca el objeto en el montículo. Para estos tipos, Java vuelve al enfoque de C y C++. Es decir, en vez de crear la variable utilizando **new**, se crea una variable "automática" que *no es una referencia*. La variable guarda el valor, y se coloca en la pila para que sea más eficiente.

Java determina el tamaño de cada tipo primitivo. Estos tamaños no varían de una plataforma a otra como ocurre en la mayoría de los lenguajes. La invariabilidad de tamaño es una de las razones por las que Java es tan llevadero.

Tipo primitivo	Tamaño	Mínimo	Máximo	Tipo de envoltura
boolean	–	–	–	**Boolean**
char	16 bits	Unicode 0	Unicode 2^{16}-1	**Character**
byte	8 bits	-1^{28}	+127	**Byte**
short	16 bits	-2^{15}	$+2^{15}-1$	**Short**
int	32 bits	-2^{31}	$+2^{31}-1$	**Integer**
long	64 bits	-2^{63}	$+2^{63}-1$	**Long**
float	32 bits	IEEE754	IEEE754	**Float**
double	64 bits	IEEE754	IEEE754	**Double**
void	–	–	–	**Void**

Todos los tipos numéricos tienen signo, de forma que es inútil tratar de utilizar tipos sin signo.

El tamaño del tipo **boolean** no está explícitamente definido; sólo se especifica que debe ser capaz de tomar los valores **true** o **false**.

Los tipos de datos primitivos también tienen clases "envoltura". Esto quiere decir que si se desea hacer un objeto no primitivo en el montículo para representar ese tipo primitivo, se hace uso del envoltorio asociado. Por ejemplo:

```
char c = 'x';
Character C = new Character (c);
```

O también se podría utilizar:

```
Character C = new Character('x');
```

Las razones para hacer esto se mostrarán más adelante en este capítulo.

Números de alta precisión

Java incluye dos clases para llevar a cabo aritmética de alta precisión: **BigInteger** y **BigDecimal**. Aunque estos tipos vienen a encajar en la misma categoría que las clases "envoltorio", ninguna de ellas tiene un tipo primitivo.

Ambas clases tienen métodos que proporcionan operaciones análogas que se lleven a cabo con tipos primitivos. Es decir, uno puede hacer con **BigInteger** y **BigDecimal** cualquier cosa que pueda hacer con un **int** o un **float**, simplemente utilizando llamadas a métodos en vez de operadores. Además, las operaciones serán más lentas dado que hay más elementos involucrados. Se sacrifica la velocidad en favor de la exactitud.

BigInteger soporta enteros de precisión arbitraria. Esto significa que uno puede representar valores enteros exactos de cualquier tamaño y sin perder información en las distintas operaciones.

BigDecimal es para números de coma flotante de precisión arbitraria; pueden usarse, por ejemplo, para cálculos monetarios exactos.

Para conocer los detalles de los constructores y métodos que pueden invocarse para estas dos clases, puede recurrirse a la documentación existente en línea.

Arrays en Java

Virtualmente, todos los lenguajes de programación soportan arrays. Utilizar arrays en C y C++ es peligroso porque los arrays no son sino bloques de memoria. Si un programa accede al array fuera del rango de su bloque de memoria o hace uso de la memoria antes de la inicialización (errores de programación bastante frecuentes) los resultados pueden ser impredecibles.

Una de los principales objetos de Java es la seguridad, de forma que muchos de los problemas habituales en los programadores de C y C++ no se repiten en Java. Está garantizado que un array en Java estará siempre inicializado, y que no se podrá acceder más allá de su rango. La comprobación de rangos se resuelve con una pequeña sobrecarga de memoria en cada array, además de verificar el índice en tiempo de ejecución, pero se asume que la seguridad y el incremento de productividad logrados merecen este coste.

Cuando se crea un array de objetos, se está creando realmente un array de referencias a los objetos, y cada una de éstas se inicializa automáticamente con un valor especial representado por la palabra clave **null**. Cuando Java ve un **null**, reconoce que la referencia en cuestión no está señalando ningún objeto. Debe asignarse un objeto a cada referencia antes de utilizarla, y si se intenta hacer uso de una referencia que aún vale **null**, se informará de que se ha dado un problema en tiempo de ejecución. Por consiguiente, en Java se evitan los errores típicos de los arrays.

Uno también puede crear un array de tipos primitivos. De nuevo, es el compilador el que garantiza la inicialización al poner a cero la memoria que ocupará ese array.

Se hablará del resto de arrays más detalladamente en capítulos posteriores.

Nunca es necesario destruir un objeto

En la mayoría de los lenguajes de programación, el concepto de tiempo de vida de una variable ocupa una parte importante del esfuerzo de programación. ¿Cuánto dura una variable? Si se supone que uno va a destruirla, ¿cuándo debe hacerse? La confusión relativa a la vida de las variables puede conducir a un montón de fallos, y esta sección muestra cómo Java simplifica enormemente esto al hacer el trabajo de limpieza por ti.

Ámbito

La mayoría de lenguajes procedurales tienen el concepto de *alcance* o *ámbito*. Éste determina tanto la visibilidad como la vida de los nombres definidos dentro de ese ámbito. En C, C++ y Java, el ámbito se determina por la ubicación de llaves **{}**. Así, por ejemplo:

```
{
	int x = 12;
	/* sólo x disponible */
	{
		int q = 96;
		/* tanto x como q están disponibles */
	}
	/* sólo x disponible */
	/* q está "fuera del ámbito o alcance" */
}
```

Una variable definida dentro de un ámbito solamente está disponible hasta que finalice su ámbito.

Las tabulaciones hacen que el código Java sea más fácil de leer. Dado que Java es un lenguaje de formato libre, los espacios extra, tabuladores y retornos de carro no afectan al programa resultante.

Fíjese que uno *no puede* hacer lo siguiente, incluso aunque sea legal en C y C++:

```
{
	int x = 12;
	{
		int x = 96; /* ilegal */
	}
}
```

El compilador comunicará que la variable **x** ya ha sido definida. Por consiguiente, la capacidad de C y C++ para "esconder" una variable de un ámbito mayor no está permitida, ya que los diseñadores de Java pensaron que conducía a programas confusos.

Ámbito de los objetos

Los objetos en Java no tienen la misma vida que los tipos primitivos. Cuando se crea un objeto Java haciendo uso de **new**, éste perdura hasta el final del ámbito. Por consiguiente, si se escribe:

```
{
      String s = new String (''un string'');
}     /* Fin del ámbito */
```

la referencia **s** desaparece al final del ámbito. Sin embargo, el objeto **String** al que apunta **s** sigue ocupando memoria. En este fragmento de código, no hay forma de acceder al objeto, pues la única referencia al mismo se encuentra fuera del ámbito. En capítulos posteriores se verá cómo puede pasarse la referencia al objeto, y duplicarla durante el curso de un programa.

Resulta que, dado que los objetos creados con **new** se mantienen durante tanto tiempo como se desee, en Java desaparecen un montón de posibles problemas propios de C++. Los problemas mayores parecen darse en C++ puesto que uno no recibe ningún tipo de ayuda del lenguaje para asegurarse de que los objetos estén disponibles cuando sean necesarios. Y lo que es aún más importante, en C++ uno debe asegurarse de destruir los objetos cuando se ha acabado con ellos.

Esto nos conduce a una cuestión interesante. Si Java deja los objetos vivos por ahí, ¿qué evita que se llene la memoria provocando que se detenga la ejecución del programa? Éste es exactamente el tipo de problema que ocurriría en C++. Es en este punto en el que ocurren un montón de cosas "mágicas". Java tiene un *recolector de basura*, que recorre todos los objetos que fueron creados con **new** y averigua cuáles no serán referenciados más. Posteriormente, libera la memoria de los que han dejado de ser referenciados, de forma que la memoria pueda ser utilizada por otros objetos. Esto quiere decir que no es necesario que uno se preocupe de reivindicar ninguna memoria. Simplemente se crean objetos, y cuando posteriormente dejan de ser necesarios, desaparecen por sí mismos. Esto elimina cierta clase de problemas de programación: el denominado "agujero de memoria", que se da cuando a un programador se le olvida liberar memoria.

Crear nuevos tipos de datos: clases

Si todo es un objeto, ¿qué determina qué apariencia tiene y cómo se comporta cada clase de objetos? O dicho de otra forma, ¿qué establece el *tipo* de un objeto? Uno podría esperar que haya una palabra clave "type", lo cual ciertamente hubiera tenido sentido. Sin embargo, históricamente, la mayoría de lenguajes orientados a objetos han hecho uso de la palabra clave **class** para indicar "Voy a decirte qué apariencia tiene un nuevo tipo de objeto". La palabra clave **class** (que se utilizará tanto

que no se pondrá en negrita a lo largo del presente libro) siempre va seguida del nombre del nuevo tipo. Por ejemplo:

```
class UnNombreDeTipo { /* Aquí va el cuerpo de la clase */ }
```

Esto introduce un nuevo tipo, de forma que ahora es posible crear un objeto de este tipo haciendo uso de la palabra clave **new**:

```
UnNombreDeTipo u = new UnNombreDeTipo ();
```

En **UnNombreDeTipo**, el cuerpo de la clase sólo consiste en un comentario (los asteriscos, las barras inclinadas y lo que hay dentro, que se discutirán más adelante en este capítulo), con lo que no hay demasiado que hacer con él. De hecho, uno no puede indicar que se haga mucho de nada (es decir, no se le puede mandar ningún mensaje interesante) hasta que se definan métodos para ella.

Campos y métodos

Cuando se define una clase (y todo lo que se hace en Java es definir clases, se hacen objetos de esas clases y se envían mensajes a esos objetos), es posible poner dos tipos de elementos en la nueva clase: datos miembros (denominados generalmente *campos*), y funciones miembros (típicamente llamados *métodos*). Un dato miembro es un objeto de cualquier tipo con el que te puedes comunicar a través de su referencia. También puede ser algún tipo primitivo (que no sea una referencia). Si es una referencia a un objeto, hay que inicializar esa referencia para conectarla a algún objeto real (utilizando **new**, como se ha visto antes) en una función especial denominada *constructor* (descrita completamente en el Capítulo 4). Si se trata de un tipo primitivo es posible inicializarla directamente en el momento de definir la clase (como se verá después, también es posible inicializar las referencias en este punto de la definición).

Cada objeto mantiene el espacio de almacenamiento necesario para todos sus datos miembro; éstos no son compartido con otros objetos. He aquí un ejemplo de una clase y algunos de sus datos miembros:

```
class SoloDatos {
      int i;
      float f;
      boolean b;
}
```

Esta clase *no hace* nada, pero es posible crear un objeto:

```
SoloDatos s = new SoloDatos();
```

Es posible asignar valores a los datos miembros, pero primero es necesario saber cómo hacer referencia a un miembro de un objeto. Esto se logra escribiendo el nombre de la referencia al objeto, seguido de un punto, y a continuación el nombre del miembro del objeto:

```
ReferenciaAObjeto.miembro
```

Por ejemplo:

```
s.i = 47;
s.f. = 1.1f;
f.b = false;
```

También es posible que un objeto pueda contener otros datos que se quieran modificar. Para ello, hay que seguir "conectando los puntos". Por ejemplo:

```
miAvion.tanqueIzquierdo.capacidad = 100;
```

La clase **SoloDatos** no puede hacer nada que no sea guardar datos porque no tiene funciones miembro (métodos). Para entender cómo funcionan los métodos, es necesario entender los *parámetros* y *valores de retorno*, que se describirán en breve.

Valores por defecto para los miembros primitivos

Cuando un tipo de datos primitivo es un miembro de una clase, se garantiza que tenga un valor por defecto siempre que no se inicialice:

Tipo primitivo	Valor por defecto
boolean	false
char	'\u0000'(null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Debe destacarse que los valores por defecto son los que Java garantiza cuando se usa la variable *como miembro de una clase*. Esto asegura que las variables miembro de tipos primitivos siempre serán inicializadas (algo que no ocurre en C++), reduciendo una fuente de errores. Sin embargo, este valor inicial puede no ser correcto o incluso legal dentro del programa concreto en el que se esté trabajando. Es mejor inicializar siempre todas las variables explícitamente.

Esta garantía no se aplica a las variables "locales" —aquellas que no sean campos de clases. Por consiguiente, si dentro de una definición de función se tiene:

```
int x;
```

Entonces **x** tomará algún valor arbitrario (como en C y C++); no se inicializará automáticamente a cero. Cada uno es responsable de asignar un valor apropiado a la variable **x** antes de usarla. Si uno se olvida, Java seguro que será mejor que C++: se recibirá un error en tiempo de compilación indicando que la variable debería haber sido inicializada. (Muchos compiladores de C++ advertirán sobre variables sin inicializar, pero en Java éstos se presentarán como errores.)

Métodos, parámetros y valores de retorno

Hasta ahora, el término *función* se ha utilizado para describir una subrutina con nombre. El término que se ha usado más frecuentemente en Java es *método*, al ser "una manera de hacer algo". Si se desea, es posible seguir pensando en funciones. Verdaderamente sólo hay una diferencia sintáctica, pero de ahora en adelante se usará el término "método" en lugar del término "función".

Los métodos en Java determinan los mensajes que puede recibir un objeto. En esta sección se aprenderá lo simple que es definir un método.

Las partes fundamentales de un método son su nombre, sus parámetros, el tipo de retorno y el cuerpo. He aquí su forma básica:

```
tipoRetorno nombreMetodo ( /* lista de parámetros */ ) {
     /* Cuerpo del método */
}
```

El tipo de retorno es el tipo del valor que surge del método tras ser invocado. La lista de parámetros indica los tipos y nombres de las informaciones que es necesario pasar a ese método. Cada método se identifica unívocamente mediante el nombre del método y la lista de parámetros.

En Java los métodos pueden crearse como parte de una clase. Es posible que un método pueda ser invocado sólo por un objeto[2], y ese objeto debe ser capaz de llevar a cabo esa llamada al método. Si se invoca erróneamente a un método de un objeto, se generará un error en tiempo de compilación. Se invoca a un método de un objeto escribiendo el nombre del objeto seguido de un punto y el nombre del método con su lista de argumentos, como: **nombreObjeto.nombreMetodo(arg1, arg2, arg3)**. Por ejemplo, si se tiene un método **f( )** que no recibe ningún parámetro y devuelve un dato de tipo **int**, y si se tiene un objeto **a** para el que puede invocarse a **f( )**, es posible escribir:

```
int x = a.f();
```

El tipo del valor de retorno debe ser compatible con el tipo de **x**.

---

[2] Los métodos **static**, que se verán más adelante, pueden ser invocados *por la clase*, sin necesidad de un objeto.

Este acto de invocar a un método suele denominarse ***envío de un mensaje a un objeto***. En el ejemplo de arriba, el mensaje es **f( )** y el objeto es **a**. La programación orientada a objetos suele resumirse como un simple "envío de mensajes a objetos".

La lista de parámetros

La lista de parámetros de un método especifica la información que se le pasa. Como puede adivinarse, esta información —como todo lo demás en Java— tiene forma de objetos. Por tanto, lo que hay que especificar en la lista de parámetros son los tipos de objetos a pasar y el nombre a utilizar en cada uno. Como en cualquier situación en Java en la que parece que se estén manipulando directamente objetos, se están pasando referencias[3]. El tipo de referencia, sin embargo, tiene que ser correcto. Si se supone, por ejemplo, que un parámetro debe ser un **String**, lo que se le pase debe ser una cadena de caracteres.

Consideremos un método que reciba como parámetro un **String**, cuya definición, que debe ser ubicada dentro de la definición de la clase para que sea compilada, puede ser la siguiente:

```
int almacenamiento (String s) {
     return s.length () * 2;
}
```

Este método dice cuántos bytes son necesarios para almacenar la información de un **String** en particular (cada **carácter** de una **cadena** tiene 16 bits, o 2 bytes para soportar caracteres Unicode). El parámetro **s** es de tipo **String**. Una vez que se pasa **s** al método, es posible tratarlo como a cualquier otro objeto (se le pueden enviar mensajes). Aquí se invoca al método **length( )**, que es uno de los métodos para **String**; devuelve el número de caracteres que tiene la cadena.

También es posible ver el uso de la palabra clave **return**, que hace dos cosas. Primero, quiere decir, "abandona el método, que ya hemos acabado". En segundo lugar, si el método produce un valor, ese valor se ubica justo después de la sentencia **return**. En este caso, el valor de retorno se produce al evaluar la expresión **s.length( )*2**.

Se puede devolver el tipo que se desee, pero si no se desea devolver nada, hay que indicar que el método devuelve **void**. He aquí algunos ejemplos:

```
boolean indicador() { return true; }
float naturalLogBase() { return 2.718f; }
void nada() { return; }
void nada2() {}
```

Cuando el tipo de retorno es **void**, se utiliza la palabra clave **return** sólo para salir del método, y es, por consiguiente, innecesaria cuando se llega al final del mismo. Es posible salir de un método en cualquier punto, pero si se te da un valor de retorno distinto de **void**, el compilador te obligará (me-

[3] Con la excepción habitual de los ya mencionados tipos de datos "especiales" **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double**. Normalmente se pasan objetos, lo cual verdaderamente quiere decir que se pasan referencias a objetos.

diante mensajes de error) a devolver el tipo apropiado de datos independientemente de lo que devuelvas.

En este punto, puede parecer que un programa no es más que un montón de objetos con métodos que toman otros objetos como parámetros y envían mensajes a esos otros objetos. Esto es, sin duda, mucho de lo que está ocurriendo, pero en el capítulo siguiente se verá cómo hacer el trabajo de bajo nivel detallado, tomando decisiones dentro de un método. Para este capítulo, será suficiente con el envío de mensajes.

Construcción de un programa Java

Hay bastantes aspectos que se deben comprender antes de ver el primer programa Java.

Visibilidad de los nombres

Un problema de los lenguajes de programación es el control de nombres. Si se utiliza un nombre en un módulo del programa, y otro programador utiliza el mismo nombre en otro módulo ¿cómo se distingue un nombre del otro para evitar que ambos nombres "colisionen"? En C éste es un problema particular puesto que un programa es un mar de nombres inmanejable. En las clases de C++ (en las que se basan las clases de Java) anidan funciones dentro de las clases, de manera que no pueden colisionar con nombres de funciones anidadas dentro de otras clases. Sin embargo, C++ sigue permitiendo los datos y funciones globales, por lo que las colisiones siguen siendo posibles. Para solucionar este problema, C++ introdujo los *espacios de nombres* utilizando palabras clave adicionales.

Java pudo evitar todo esto siguiendo un nuevo enfoque. Para producir un nombre no ambiguo para una biblioteca, el identificador utilizado no difiere mucho de un nombre de dominio en Internet. De hecho, los creadores de Java utilizaron los nombres de dominio de Internet a la inversa, dado que es posible garantizar que éstos sean únicos. Dado que mi nombre de dominio es **BruceEckel.com**, mi biblioteca de utilidad de **manías** debería llamarse **com.bruceEckel.utilidad.manias**. Una vez que se da la vuelta al nombre de dominio, los nombres supuestamente representan subdirectorios.

En Java 1.0 y 1.1, las extensiones de dominio **com**, **edu**, **org**, **net**, etc. se ponían en mayúsculas por convención, de forma que la biblioteca aparecería como **COM.bruceEckel.utilidad.manias.** Sin embargo, a mitad de camino del desarrollo de Java 2, se descubrió que esto causaba problemas, por lo que de ahora en adelante se utilizarán minúsculas para todas las letras de los nombres de paquetes.

Este mecanismo hace posible que todos sus ficheros residan automáticamente en sus propios espacios de nombres, y cada clase de un fichero debe tener un identificador único. Por tanto, uno no necesita aprender ninguna característica especial del lenguaje para resolver el problema —el lenguaje lo hace por nosotros.

Utilización de otros componentes

Cada vez que se desee usar una clase predefinida en un programa, el compilador debe saber dónde localizarla. Por supuesto, la clase podría existir ya en el mismo fichero de código fuente que la está invocando. En ese caso, se puede usar simplemente la clase —incluso si la clase no se define hasta más adelante dentro del archivo. Java elimina el problema de las "referencias hacia delante" de forma que no hay que pensar en ellos.

¿Qué hay de las clases que ya existen en cualquier otro archivo? Uno podría pensar que el compilador debería ser lo suficientemente inteligente como para localizarlo por sí mismo, pero hay un problema. Imagínese que se quiere usar una clase de un nombre determinado, pero existe más de una definición de esa clase (y presumiblemente se trata de definiciones distintas). O peor, imagine que se está escribiendo un programa, y a medida que se está construyendo se añade una nueva clase a la biblioteca cuyo nombre choca con el de alguna clase ya existente.

Para resolver este problema, debe eliminarse cualquier ambigüedad potencial. Esto se logra diciéndole al compilador de Java exactamente qué clases se quieren utilizar mediante la palabra clave **import**. Esta palabra clave dice al compilador que traiga un *paquete*, que es una biblioteca de clases (en otros lenguajes, una biblioteca podría consistir en funciones y datos además de clases, pero debe recordarse que en Java todo código debe escribirse dentro de una clase).

La mayoría de las veces se utilizarán componentes de las bibliotecas de Java estándar que vienen con el propio compilador. Con ellas, no hay que preocuparse de los nombres de dominio largos y dados la vuelta; uno simplemente dice, por ejemplo:

```
import java.util.ArrayList;
```

para indicar al compilador que se desea utilizar la clase **ArrayList** de Java. Sin embargo, **util** contiene bastantes clases y uno podría querer utilizar varias de ellas sin tener que declararlas todas explícitamente. Esto se logra sencillamente utilizando el '*' que hace las veces de comodín:

```
import java.util.*;
```

Es más común importar una colección de clases de esta forma que importar las clases individualmente.

La palabra clave **static**

Generalmente, al crear una clase se está describiendo qué apariencia tienen sus objetos y cómo se comportan. No se tiene nada hasta crear un objeto de esa clase con **new**, momento en el que se crea el espacio de almacenamiento y los métodos pasan a estar disponibles.

Pero hay dos situaciones en las que este enfoque no es suficiente. Una es cuando se desea tener solamente un fragmento de espacio de almacenamiento para una parte concreta de datos, independientemente de cuántos objetos se creen, o incluso aunque no se cree ninguno. La otra es si se necesita un método que no esté asociado con ningún objeto particular de esa clase. Es decir, se necesita un método al que se pueda invocar incluso si no se ha creado ningún objeto. Ambos efec-

tos se pueden lograr con la palabra clave **estático**. Al decir que algo es **estático** se está indicando que el dato o método no está atado a ninguna instancia de objeto de esa clase en particular. Por ello, incluso si nunca se creó un objeto de esa clase se puede invocar a un método **estático** o acceder a un fragmento de datos **estático**. Con los métodos y datos ordinarios no **estático**, es necesario crear un objeto y utilizarlo para acceder al dato o método, dado que los datos y métodos no **estático** deben conocer el objeto particular con el que está trabajando. Por supuesto, dado que los métodos **estático** no precisan de la creación de ningún objeto, no pueden acceder *directamente* a miembros o métodos no **estático** simplemente invocando a esos otros miembros sin referirse a un objeto con nombre (dado que los miembros y objetos no **estático** deben estar unidos a un objeto en particular).

Algunos lenguajes orientados a objetos utilizan los términos *datos a nivel de clase* y *métodos a nivel de clase*, para indicar que los datos y métodos solamente existen para la clase, y no para un objeto particular de la clase. En ocasiones, estos términos también se usan en los textos.

Para declarar un dato o un miembro a nivel de clase **estático**, basta con colocar la palabra clave **estático** antes de la definición. Por ejemplo, el siguiente fragmento produce un miembro de datos **estáticos** y lo inicializa:

```
class PruebaEstatica {
      static int i = 47;
}
```

Ahora, incluso si se construyen dos objetos de Tipo **PruebaEstatica**, sólo habrá un espacio de almacenamiento para **PruebaEstatica.i**. Ambos objetos compartirán la misma **i**. Considérese:

```
PruebaEstatica st1 = new PruebaEstatica();
PruebaEstatica st2 = new PruebaEstatica();
```

En este momento, tanto **st1.i** como **st2.i** tienen el valor 47, puesto que se refieren al mismo espacio de memoria.

Hay dos maneras de referirse a una variable **estática**. Como se indicó más arriba, es posible nombrarlas a través de un objeto, diciendo, por ejemplo, **st2.i**. También es posible referirse a ella directamente a través de su nombre de clase, algo que no se puede hacer con miembros no estáticos (ésta es la manera preferida de referirse a una variable **estática** puesto que pone especial énfasis en la naturaleza **estática** de esa variable).

```
PruebaEstatica.i++:
```

El operador ++ incrementa la variable. En este momento, tanto **st1.i** como **st2.i** valdrán 48.

Algo similar se aplica a los métodos estáticos. Es posible hacer referencia a ellos, bien a través de un objeto especificado al igual, que ocurre con cualquier método, o bien con la sintaxis adicional **NombreClase.método( )**. Un método estático se define de manera semejante:

```
class FunEstatico {
      static void incr() {PruebaEstatica.i++; }
}
```

Puede observarse que el método **incr( )** de **FunEstatico** incrementa la variable **estática i**. Se puede invocar a **incr( )** de la manea típica, a través de un objeto:

```
FunEstatico sf = new FunEstatico();
sf.incr();
```

O, dado que **incr()** es un método estático, es posible invocarlo directamente a través de la clase:

```
FunEstatico.incr();
```

Mientras que **static** al ser aplicado a un miembro de datos, cambia definitivamente la manera de crear los datos (uno por cada clase en vez de uno por cada objeto no **estático**), al aplicarse a un método, su efecto no es tan drástico. Un uso importante de **estático** para los métodos es permitir invocar a un método sin tener que crear un objeto. Esto, como se verá, es esencial en la definición del método **main( )**, que es el punto de entrada para la ejecución de la aplicación.

Como cualquier método, un método **estático** puede crear o utilizar objetos con nombre de su propio tipo, de forma que un método **estático** se usa a menudo como un "pastor de ovejas" para un conjunto de instancias de su mismo tipo.

Tu primer programa Java

Finalmente, he aquí el programa[4]. Empieza imprimiendo una cadena de caracteres y posteriormente la fecha, haciendo uso de la clase **Date**, contenida en la biblioteca estándar de Java. Hay que tener en cuenta que se introduce un estilo de comentarios adicional: el '//', que permite insertar un comentario hasta el final de la línea:

```
// HolaFecha.java
import java.util.*;

public class HolaFecha {
    public static void main(String[] args) {
        System.out.println ("Hola, hoy es: ");
        System.out.println (new Date());
    }
}
```

Al principio de cada fichero de programa es necesario poner la sentencia **import** para incluir cualquier clase adicional que se necesite para el código contenido en ese fichero. Nótese que digo "adi-

[4] Algunos entornos de programación irán sacando programas en la pantalla, y luego los cerrarán antes de que uno tenga siquiera opción a ver los resultados. Para detener la salida, se puede escribir el siguiente fragmento de código al final de la función **main ( )**;

```
try {
  System.in.read();
  } catch(Exception e) {}
```

Esto hará que la salida se detenga hasta presionar "Intro" (o cualquier otra tecla). Este código implica algunos conceptos que no se verán hasta mucho más adelante, por lo que todavía no lo podemos entender, aunque el truco es válido igualmente.

cional"; se debe a que hay una cierta biblioteca de clases que se carga automáticamente en todos los ficheros Java: la **java.lang**. Arranque su navegador web y eche un vistazo a la documentación de Sun (si no la ha bajado de *java.sun.com* o no ha instalado la documentación de algún otro modo, será mejor hacerlo ahora). Si se echa un vistazo a la lista de paquetes, se verán todas las bibliotecas de clases que incluye Java. Si se selecciona **java.lang** aparecerá una lista de todas las clases que forman parte de esa biblioteca. Dado que **java.lang** está incluida implícitamente en todos los archivos de código Java, todas estas clases ya estarán disponibles. En **java.lang** no hay ninguna clase **Date**, lo que significa que será necesario importarla de alguna otra biblioteca. Si se desconoce en qué biblioteca en particular está una clase, o si se quieren ver todas las clases, es posible seleccionar "Tree" en la documentación de Java. En ese momento es posible encontrar todas y cada una de las clases que vienen con Java. Después, es posible hacer uso de la función "buscar" del navegador para encontrar **Date**. Al hacerlo, se verá que está listada como **java.util.Date**, lo que quiere decir que se encuentra en la biblioteca **util**, y que es necesario importar **java.util.*** para poder usar **Date**.

Si se vuelve al principio, se selecciona **java.lang** y después **System**, se verá que la clase **System** tiene varios campos, y si se selecciona **out**, se descubrirá que es un objeto **estático PrintStream**. Dado que es **estático**, no es necesario crear ningún objeto. El objeto **out** siempre está ahí y se puede usar directamente. Lo que se hace con el objeto **out** está determinado por su tipo: **PrintStream**. La descripción de este objeto se muestra, convenientemente, a través de un hipervínculo, por lo que si se hace clic en él se verá una lista de todos los métodos de **PrintStream** a los que se puede invocar. Hay unos cuantos, y se irán viendo según avancemos en la lectura del libro. Por ahora, todo lo que nos interesa es **println( )**, que significa "escribe lo que te estoy dando y finaliza con un retorno de carro". Por consiguiente, en cualquier programa Java que uno escriba se puede decir **System.out.println("cosas")** cuando se desee para escribir algo en la consola.

El nombre de la clase es el mismo que el nombre del archivo. Cuando se está creando un programa independiente como éste, una de las clases del archivo tiene que tener el mismo nombre que el archivo. (El compilador se queja si no se hace así.) Esa clase debe contener un método llamado **main( )**, de la forma:

```
public static void main(String[] args) {
```

La palabra clave **public** quiere decir que el método estará disponible para todo el mundo (como se describe en el Capítulo 5). El parámetro del método **main( )** es un array de objetos **String**. Este programa no usará **args**, pero el compilador Java obliga a que esté presente, pues son los que mantienen los parámetros que se invoquen en la línea de comandos.

La línea que muestra la fecha es bastante interesante:

```
System.out.println(new Date());
```

Considérese su argumento: se está creando un objeto **Date** simplemente para enviar su valor a **println**. Tan pronto como haya acabado esta sentencia, ese **Date** deja de ser necesario, y en cualquier momento aparecerá el recolector de basura y se lo llevará. Uno no tiene por qué preocuparse de limpiarlo.

Compilación y ejecución

Para compilar y ejecutar este programa, y todos los demás programas de este libro, es necesario disponer, en primer lugar, de un entorno de programación Java. Hay bastantes entornos de desarrollo de terceros, pero en este libro asumiremos que se está usando el JDK de Sun, que es gratuito. Si se está utilizando otro sistema de desarrollo, será necesario echar un vistazo a la documentación de ese sistema para determinar cómo se compilan y ejecutan los programas.

Conéctese a Internet y acceda a *java.sun.com*. Ahí encontrará información y enlaces que muestran cómo descargar e instalar el JDK para cada plataforma en particular.

Una vez que se ha instalado el JDK, y una vez que se ha establecido la información de *path* en el computador, para que pueda encontrar **javac** y **java**, se puede descargar e instalar el código fuente de este libro (que se encuentra en el CD ROM que viene con el libro, o en *http.//www.BruceEckel.com*). Al hacerlo, se creará un subdirectorio para cada capítulo del libro. Al ir al subdirectorio **c02** y escribir:

```
javac HolaFecha.java
```

no se obtendrá ninguna respuesta. Si se obtiene algún mensaje de error, se debe a que no se ha instalado el JDK correctamente, por lo que será necesario ir investigando los problemas que se muestren.

Por otro lado, si simplemente ha vuelto a aparecer el *prompt* del intérprete de comandos, basta con teclear:

```
java HolaFecha
```

y se obtendrá como salida el mensaje y la fecha.

Éste es el proceso a seguir para compilar y ejecutar cada uno de los programas de este libro. Sin embargo, se verá que el código fuente de este libro también tiene un archivo denominado **makefile** en cada capítulo, que contiene comandos "make" para construir automáticamente los archivos de ese capítulo. Puede verse la página web del libro en *http://www.BruceEckel.com* para ver los detalles de uso de los *makefiles*.

Comentarios y documentación empotrada

Hay dos tipos de comentarios en Java. El primero es el estilo de comentarios tradicional de C, que fue heredado por C++. Estos comentarios comienzan por /* y pueden extenderse incluso a lo largo de varias líneas hasta encontrar */. Téngase en cuenta que muchos programadores comienzan cada línea de un comentario continuo por el signo *, por lo que a menudo se verá:

```
/*    Esto es un comentario
*     que se extiende
*     a lo largo de varias líneas
*/
```

Hay que recordar, sin embargo, que todo lo que esté entre /* y */ se ignora, por lo que no hay ninguna diferencia con decir:

```
/* Éste es un comentario que
se extiende a lo largo de varias líneas */
```

La segunda forma de hacer comentarios viene de C++. Se trata del comentario en una sola línea que comienza por // y continúa hasta el final de la línea. Este tipo de comentario es muy conveniente y se utiliza muy frecuentemente debido a su facilidad de uso. Uno no tiene que buscar por el teclado donde está el / y el * (basta con pulsar dos veces la misma tecla), y no es necesario cerrar el comentario, por lo que a menudo se verá:

```
// esto es un comentario en una sola línea
```

Documentación en forma de comentarios

Una de las partes más interesantes del lenguaje Java es que los diseñadores no sólo tuvieron en cuenta que la escritura de código era la única actividad importante —sino que también pensaron en la documentación del código. Probablemente el mayor problema a la hora de documentar el código es el mantenimiento de esa documentación. Si la documentación y el código están separados, cambiar la documentación cada vez que se cambia el código se convierte en un problema. La solución parece bastante simple: unir el código a la documentación. La forma más fácil de hacer esto es poner todo en el mismo archivo. Para completar la estampa, sin embargo, es necesaria alguna sintaxis especial de comentarios para marcarlos como documentación especial, y una herramienta para extraer esos comentarios y ponerlos en la forma adecuada.

La herramienta para extraer los comentarios se denomina *javadoc*. Utiliza parte de la tecnología del compilador de Java para buscar etiquetas de comentario especiales que uno incluye en sus programas. No sólo extrae la información marcada por esas etiquetas, sino que también extrae el nombre de la clase o del método al que se adjunta el comentario. De esta manera es posible invertir la mínima cantidad de trabajo para generar una decente documentación para los programas.

La salida de javadoc es un archivo HTML que puede visualizarse a través del navegador Web. Esta herramienta permite la creación y mantenimiento de un único archivo fuente y genera automáticamente documentación útil. Gracias a javadoc se tiene incluso un estándar para la creación de documentación, tan sencillo que se puede incluso esperar o solicitar documentación con todas las bibliotecas Java.

Sintaxis

Todos los comandos de javadoc se dan únicamente en comentarios /**. Estos comentarios acaban, como siempre, con */. Hay dos formas principales de usar javadoc: empotrar HTML, o utilizar "etiquetas doc". Las etiquetas doc son comandos que comienzan por '@' y se sitúan al principio de una línea de comentarios (en la que se ignora un posible primer '*').

Hay tres "tipos" de documentación en forma de comentarios, que se corresponden con el elemento al que precede el comentario: una clase, una variable o un método. Es decir, el comentario relativo

a una clase aparece justo antes de la definición de la misma; el comentario relativo a una variable precede siempre a la definición de la variable, y un comentario de un método aparece inmediatamente antes de la definición de un método. Un simple ejemplo:

```
/** Un comentario de clase */
public class PruebaDoc {
      /** Un comentario de una variable */
      public int i;
      /** Un comentario de un método */
      public void f() {}
}
```

Nótese que javadoc procesará la documentación en forma de comentarios sólo de miembros **public** y **protected**. Los comentarios para miembros **private** y "friendly" (véase Capítulo 5) se ignoran, no mostrándose ninguna salida (sin embargo es posible usar el modificador **—private** para incluir los miembros **privados**). Esto tiene sentido, dado que sólo los miembros **públicos** y **protegidos** son visibles fuera del objeto, que será lo que constituya la perspectiva del programador cliente. Sin embargo, la salida incluirá todos los comentarios de la **clase**.

La salida del código anterior es un archivo HTML que tiene el mismo formato estándar que toda la documentación Java, de forma que los usuarios se sientan cómodos con el formato y puedan navegar de manera sencilla a través de sus clases. Merece la pena introducir estos códigos, pasarlos a través de javadoc y observar el fichero HTML resultante para ver los resultados.

HTML empotrado

Javadoc pasa comandos HTML al documento HTML generado. Esto permite un uso total de HTML; sin embargo, el motivo principal es permitir dar formato al código, como:

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

También puede usarse HTML como se haría en cualquier otro documento web para dar formato al propio texto de las descripciones:

```
/**
* Uno puede <em>incluso</em> insertar una lista:
* <ol>
* <li> Elemento uno
* <li> Elemento dos
* <li> Elemento tres
* </ol>
*/
```

Nótese que dentro de los comentarios de documentación, los asteriscos que aparezcan al principio de las líneas serán desechados por javadoc, junto con los espacios adicionales a éstos. Javadoc vuelve a dar formato a todo adaptándolo a la apariencia estándar de la documentación. No deben utilizarse encabezados como <**hl**> o <**hr**> como HTML empotrado porque javadoc inserta sus propios encabezados y éstos interferirían con ellos.

Todos los tipos de documentación en comentarios —de clases, variables y métodos— soportan HTML empotrado.

@see: referencias a otras clases

Los tres tipos de comentarios de documentación (de clase, variable y métodos) pueden contener etiquetas **@see**, que permiten hacer referencia a la documentación de otras clases. Javadoc generará HTML con las etiquetas **@see** en forma de vínculos a la otra documentación. Las formas son:

```
@see nombredeclase
@see nombredeclase-totalmente-cualificada
@see nombredeclase-totalmente-cualificada#nombre-metodo
```

Cada una añade un hipervínculo "Ver también" a la documentación generada. Javadoc no comprobará los hipervínculos que se le proporcionen para asegurarse de que sean válidos.

Etiquetas de documentación de clases

Junto con el HTML empotrado y las referencias **@see**, la documentación de clases puede incluir etiquetas de información de la versión y del nombre del autor. La documentación de clases también puede usarse para las *interfaces* (véase Capítulo 8).

@versión

Es de la forma:

```
@versión información-de-versión
```

en el que **información-de-versión** es cualquier información significativa que se desee incluir. Cuando se especifica el indicador -**versión** en la línea de comandos javadoc, se invocará especialmente a la información de versión en la documentación HTML generada.

@author

Es de la forma:

```
@autor información-del-autor
```

donde la **información-del-autor** suele ser el nombre, pero podría incluir también la dirección de correo electrónico u otra información apropiada. Al activar el parámetro -author en la línea de comandos javadoc, se invocará a la información relativa al autor en la documentación HTML generada.

Se pueden tener varias etiquetas de autor, en el caso de tratarse de una lista de autores, pero éstas deben ponerse consecutivamente. Toda la información del autor se agrupará en un único párrafo en el HTML generado.

@since

Esta etiqueta permite indicar la versión del código que comenzó a utilizar una característica concreta. Se verá que aparece en la documentación para ver la versión de JDK que se está utilizando.

Etiquetas de documentación de variables

La documentación de variables solamente puede incluir HTML empotrado y referencias **@see**.

Etiquetas de documentación de métodos

Además de documentación empotrada y referencias **@see**, los métodos permiten etiquetas de documentación para los parámetros, los valores de retorno y las excepciones.

@param

Es de la forma:

```
@param  nombre-parámetro descripción
```

donde **nombre-parámetro** es el identificador de la lista de parámetros, y **descripción** es el texto que vendrá en las siguientes líneas. Se considera que la descripción ha acabado cuando se encuentra una nueva etiqueta de documentación. Se puede tener cualquier número de estas etiquetas, generalmente una por cada parámetro.

@return

Es de la forma:

```
@return descripción
```

donde **descripción** da el significado del valor de retorno. Puede ocupar varias líneas.

@throws

Las excepciones se verán en el Capítulo 10, pero sirva como adelanto que son objetos que pueden "lanzarse" fuera del método si éste falla. Aunque al invocar a un método sólo puede lanzarse una excepción, podría ocurrir que un método particular fuera capaz de producir distintos tipos de excepciones, necesitando cada una de ellas su propia descripción. Por ello, la etiqueta de excepciones es de la forma:

```
@throws nombre-de-clase-totalmente-cualificada descripción
```

donde **nombre-de-clase-totalmente-cualificada** proporciona un nombre sin ambigüedades de una clase de excepción definida en algún lugar, y **descripción** (que puede extenderse a lo largo de varias líneas) indica por qué podría levantarse este tipo particular de excepción al invocar al método.

@deprecated

Se utiliza para etiquetar aspectos que fueron mejorados. Esta etiqueta es una sugerencia para que no se utilice esa característica en particular nunca más, puesto que en algún momento del futuro puede que se elimine. Un método marcado como **@deprecated** hace que el compilador presente una advertencia cuando se use.

Ejemplo de documentación

He aquí el primer programa Java de nuevo, al que en esta ocasión se ha añadido documentación en forma de comentarios:

```
//: c02:HolaFecha.java
import java.util.*;

/** El primer ejemplo de Piensa en Java.
  *  Muestra una cadena de caracteres y la fecha de hoy.
  *  @author Bruce Eckel
  *  @author www.BruceEckel.com
  *  @version 2.0
*/
public class HolaFecha {
      /** Único punto de entrada para la clase y la aplicación
        * @param args array de cadenas de texto pasadas como
parámetros
        * @return No hay valor de retorno
        * @exception exceptions No se generarán excepciones
      */
public static void main (String[] args){
      System.out.printl("Hola, hoy es: ");
      System.out.println(new Date());
}
} ///:~
```

La primera línea del archivo utiliza mi propia técnica de poner ":" como marcador especial de la línea de comentarios que contiene el nombre del archivo fuente. Esa línea contiene la información de la trayectoria al fichero (en este caso, **c02** indica el Capítulo 2) seguido del nombre del archivo[5]. La última línea también acaba con un comentario, esta vez indicando la finalización del listado de código fuente, que permite que sea extraído automáticamente del texto de este libro y comprobado por un compilador.

---

[5] Una herramienta que he creado usando Python (ver http://www.Python.org) utiliza esta información para extraer esos ficheros de código, ponerlos en los subdirectorios apropiados y crear los "makefiles".

Estilo de codificación

El estándar no oficial de java dice que se ponga en mayúsculas la primera letra del nombre de una clase. Si el nombre de la clase consta de varias palabras, se ponen todas juntas (es decir, no se usan guiones bajos para separar los nombres) y se pone en mayúscula la primera letra de cada palabra, como por ejemplo:

```
class TodosLosColoresDelArcoiris { // ...
```

En casi todo lo demás: métodos, campos (variables miembro), y nombres de referencias a objeto, el estilo aceptado es el mismo que para las clases, con la *excepción* de que la primera letra del identificador debe ser minúscula. Por ejemplo:

```
class TodosLosColoresDelArcoiris {
      int unEnteroQueRepresentaUnColor;
      void cambiarElTonoDelColor (int nuevoTono) {
      // ...
      }
      // ...
}
```

Por supuesto, hay que recordar que un usuario tendría que teclear después todos estos nombres largos, por lo que se ruega a los programadores que lo tengan en cuenta.

El código Java de las bibliotecas de Sun también sigue la forma de apertura y cierre de las llaves que se utilizan en este libro.

Resumen

En este capítulo se ha visto lo suficiente de programación en Java como para entender cómo escribir un programa sencillo, y se ha realizado un repaso del lenguaje y algunas de sus ideas básicas. Sin embargo, los ejemplos hasta la fecha han sido de la forma "haz esto, después haz esto otro, y finalmente haz algo más". ¿Y qué ocurre si quieres que el programa presente alternativas, como "si el resultado de hacer esto es rojo, haz esto; sino, haz no se qué más?" El soporte que Java proporciona a esta actividad fundamental de programación se verá en el capítulo siguiente.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Siguiendo el ejemplo **HolaFecha.java** de este capítulo, crear un programa "Hola, mundo" que simplemente escriba esa frase. Sólo se necesita un método en la clase (la clase "main" que es la que se ejecuta al arrancar el programa). Recordar hacerla **static** e incluir la lista de parámetros, incluso aunque no se vaya a usar. Compilar el programa con **javac** y ejecutarlo

utilizando **java**. Si se utiliza un entorno de desarrollo distinto a JDK, aprender a compilar y ejecutar programas en ese entorno.

2. Encontrar los fragmentos de código involucrados en **UnNombreDeTipo** y convertirlos en un programa que se compile y ejecute.
3. Convertir los fragmentos de código de **SoloDatos** en un programa que se compile y ejecute.
4. Modificar el Ejercicio 3, de forma que los valores de los datos de **SoloDatos** se asignen e impriman en **main( )**.
5. Escribir un programa que incluya y llame al método **almacenamiento( )**, definido como fragmento de código en este capítulo.
6. Convertir los fragmentos de código de **FunEstatico** en un programa ejecutable.
7. Escribir un programa que imprima tres parámetros tomados de la línea de comandos. Para lograrlo, será necesario indexarlos en el array de **Strings** de línea de comandos.
8. Convertir el ejemplo **TodosLosColoresDelArcoiris** en un programa que se compile y ejecute.
9. Encontrar el código de la segunda versión de **HolaFecha.java**, que es el ejemplo de documentación en forma de comentarios. Ejecutar **javadoc** del fichero y observar los resultados con el navegador web.
10. Convertir **PruebaDoc** en un fichero que se compile y pasarlo por **javadoc**. Verificar la documentación resultante con el navegador web.
11. Añadir una lista de elementos HTML a la documentación del Ejercicio 10.
12. Tomar el programa del Ejercicio 10 y añadirle documentación en forma de comentarios. Extraer esta documentación en forma de comentarios a un fichero HTML utilizando **javadoc** y visualizarla con un navegador web.

3: Controlar el flujo del programa

Al igual que una criatura con sentimientos, un programa debe manipular su mundo y tomar decisiones durante su ejecución.

En Java, se manipulan objetos y datos haciendo uso de operadores, y se toman decisiones con la ejecución de sentencias de control. Java se derivó de C++, por lo que la mayoría de esas sentencias y operadores resultarán familiares a los programadores de C y C++. Java también ha añadido algunas mejoras y simplificaciones.

Si uno se encuentra un poco confuso durante este capítulo, acuda al CD ROM multimedia adjunto al libro: *Thinking in C: Foundations for Java and C++*. Contiene conferencias sonoras, diapositivas, ejercicios y soluciones diseñadas específicamente para ayudarle a adquirir familiaridad con la sintaxis de C necesaria para aprender Java.

Utilizar operadores de Java

Un operador toma uno o más parámetros y produce un nuevo valor. Los parámetros se presentan de distinta manera que en las llamadas ordinarias a métodos, pero el efecto es el mismo. Uno debería estar razonablemente cómodo con el concepto general de operador con su experiencia de programación previa. La suma (+), la resta y el menos unario (–), la multiplicación (*), la división (/), y la asignación (=) funcionan todos exactamente igual que en el resto de lenguajes de programación.

Todos los operadores producen un valor a partir de sus operandos. Además, un operador puede variar el valor de un operando. A esto se le llama *efecto lateral*. El uso más común de los operadores que modifican sus operandos es generar el efecto lateral, pero uno debería tener en cuenta que el valor producido solamente podrá ser utilizado en operadores sin efectos laterales.

Casi todos los operadores funcionan únicamente con datos primitivos. Las excepciones las constituyen "=", "==" y "!=", que funcionan con todos los objetos (y son una fuente de confusión para los objetos). Además, la clase **String** soporta "+" y "+=".

Precedencia

La precedencia de los operadores define cómo se evalúa una expresión cuando hay varios operadores en la misma. Java tiene reglas específicas que determinan el orden de evaluación. La más fácil de recordar es que la multiplicación y la división siempre se dan tras la suma y la resta. Los programadores suelen olvidar el resto de reglas de precedencia a menudo, por lo que se deberían usar paréntesis para establecer explícitamente el orden de evaluación. Por ejemplo:

```
A = X + Y - 2/2 + Z;
```

tiene un significado diferente que la misma sentencia con una agrupación particular de paréntesis:

```
A = X + ( Y - 2 ) / ( 2 + Z );
```

Asignación

La asignación se lleva a cabo con el operador =. Significa "toma el valor de la parte derecha (denominado a menudo *dvalor*) y cópialo a la parte izquierda (a menudo denominada *ivalor*"). Un ivalor es cualquier constante, variable o expresión que pueda producir un valor, pero un ivalor debe ser una variable única con nombre. (Es decir, debe haber un espacio físico en el que almacenar un valor.) Por ejemplo, es posible asignar un valor constante a una variable (**A = 4;**), pero no se puede asignar nada a un valor constante —no puede ser un ivalor. (No se puede decir **4 = A;**.)

La asignación de tipos primitivos de datos es bastante sencilla y directa. Dado que el dato primitivo alberga el valor actual y no una referencia a un objeto, cuando se asignan primitivas se copian los contenidos de un sitio a otro. Por ejemplo, si se dice **A = B** para datos primitivos, los contenidos de **B** se copian a **A**. Si después se intenta modificar **A**, lógicamente **B** no se verá alterado por esta modificación. Como programador, esto es lo que debería esperarse en la mayoría de situaciones.

Sin embargo, cuando se asignan objetos, las cosas cambian. Siempre que se manipula un objeto, lo que se está manipulando es la referencia, por lo que al hacer una asignación "de un objeto a otro" se está, de hecho, copiando una referencia de un sitio a otro. Esto significa que si se escribe **C = D** siendo ambos objetos, se acaba con que tanto **C** como **D** apuntan al objeto al que originalmente sólo apuntaba **D**. El siguiente ejemplo demuestra esta afirmación.

He aquí el ejemplo:

```
//: c03:Asignacion.java
// La asignación con objetos tiene su truco.

class Numero {
  int i;
}

public class Asignacion {
  public static void main(String[] args) {
    Numero n1 = new Numero();
    Numero n2 = new Numero();
    n1.i = 9;
    n2.i = 47;
    System.out.println("1:n1.i: " + n1.i + ", n2.i: " + n2.i);
    n1 = n2;
    System.out.println("2: n1.i: " + n1.i + ", n2.i: " + n2.i);
    n1.i = 27;
    System.out.println("3: n1.i: " + n1.i + ", n2.i: " + n2.i);
  }
} ///:~
```

La clase **Número** es sencilla, y sus dos instancias (**n1** y **n2**) se crean dentro del método **main()**. Al valor **i** de cada **Número** se le asigna un valor distinto, y posteriormente se asigna **n2** a **n1**, y se varía **n1**. En muchos lenguajes de programación se esperaría que **n1** y **n2** fuesen independientes, pero dado que se ha asignado una referencia, he aquí la salida que se obtendrá:

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

Al cambiar el objeto **n1** parece que se cambia el objeto **n2** también. Esto ocurre porque, tanto **n1,** como **n2** contienen la misma referencia, que apunta al mismo objeto. (La referencia original que estaba en **n1** que apuntaba al objeto que albergaba el valor 9 fue sobreescrita durante la asignación y, en consecuencia, se perdió; su objeto será eliminado por el recolector de basura.)

A este fenómeno se le suele denominar *uso de alias* y es una manera fundamental que tiene Java de trabajar con los objetos. Pero, ¿qué ocurre si uno no desea que se dé dicho uso de alias en este caso? Uno podría ir más allá con la asignación y decir:

```
n1.i = n2.i;
```

Esto mantiene los dos objetos separados en vez de desechar uno y vincular **n1** y **n2** al mismo objeto, pero pronto nos damos cuenta que manipular los campos de dentro de los objetos es complicado y atenta contra los buenos principios de diseño orientado a objetos. Este asunto no es trivial, por lo que se deja para el Apéndice A, dedicado al uso de alias. Mientras tanto, se debe recordar que la asignación de objetos puede traer sorpresas.

Uso de alias durante llamadas a métodos

También puede darse uso de alias cuando se pasa un objeto a un método:

```
//: c03:PasarObjeto.java
// Pasar objetos a métodos puede no ser aquello a lo que uno está
// acostumbrado.
class Carta {
  char c;
}

public class PasarObjecto {
  static void f(Carta y) {
    y.c = 'z';
  }
  public static void main(String[] args) {
    Carta x = new Carta();
    x.c = 'a';
    System.out.println("1: x.c: " + x.c);
    f(x);
    System.out.printl("2:x.c: " + x.c);
```

```
  }
} ///:~
```

En muchos lenguajes de programación el método **f( )** parecería estar haciendo una copia de su argumento **Carta** y dentro del ámbito del método. Pero una vez más, se está pasando una referencia, por lo que la línea:

```
y.c = 'z';
```

está, de hecho, cambiando el objeto fuera de **f( )**. La salida tiene el aspecto siguiente:

```
1: x.c: a
2: x.c: z
```

El uso de alias y su solución son un aspecto complejo, y aunque uno debe esperar al Apéndice A para tener todas las respuestas, hay que ser consciente de este problema desde este momento, de forma que podamos estar atentos y no caer en la trampa.

Operadores matemáticos

Los operadores matemáticos básicos son los mismos que los disponibles en la mayoría de lenguajes de programación: suma (+), resta (-), división (/), multiplicación (*) y módulo (%, que devuelve el resto de una división entera). La división entera trunca, en vez de redondear, el resultado.

Java también utiliza una notación abreviada para realizar una operación y llevar a cabo una asignación simultáneamente. Este conjunto de operaciones se representa mediante un operador seguido del signo igual, y es consistente con todos los operadores del lenguaje (cuando tenga sentido). Por ejemplo, para añadir 4 a la variable **x** y asignar el resultado a **x** puede usarse: **x+=4**.

El siguiente ejemplo muestra el uso de los operadores matemáticos:

```
//: c03:OperadoresMatematicos.java
// Demuestra los operadores matemáticos
import java.util.*;

public class OperadoresMatematicos {
  // Crear un atajo para ahorrar teclear:
  static void visualizar(String s) {
    System.out.println(s);
  }
  // Atajo para visualizar un string y un entero:
  static void pInt (String s, int i) {
    visualizar(s + " = " + i);
  }
  // Atajo para visualizar una cadena de caracteres y un float:
  static void pFlt(String s, float f) {
    visualizar(s + " = " + f);
  }
```

```
  public static void main(String [] args) {
    // Crear un generador de números aleatorios
    // El generador se alimentará por defecto de la hora actual:
    Random aleatorio = new Random();
    int i, j, k;
    // '%' limita el valor a 99:
    j = aleatorio.nextInt() % 100;
    k = aleatorio.nextInt() %100;
    pInt ("j",j); pInt("k",k);
    i = j + k; pInt("j + k", i);
    i = j - k; pInt("j - k", i);
    i = k / j; pInt("k / j", i);
    i = k *j; pInt("k * j", i);
    i = k % j; pInt("k % j", i);
    j %= k; pInt("j %= k", j);
    // Pruebas de números de coma flotante:
    float u,v,w;  // Se aplica también a doubles
    v = aleatorio.nextFloat();
    w = aleatorio.nextFloat();
    pFlt("v", v); pFlt("w", w);
    u = v + w; pFlt("v + w", u);
    u = v - w; pFlt("v - w", u);
    u = v * w; pFlt("v * w", u);
    u = v / w; pFlt("v / w", u);
    // Lo siguiente funciona también para char, byte
    // short, int, long, y double:
    u += v; pFlt("u += v", u);
    u -= v; pFlt("u -= v", u);
    u *= v; pFlt("u *= v", u);
    u /= v; pFlt("u /= v", u);
  }
} ///:~
```

Lo primero que se verán serán los métodos relacionados con la visualización por pantalla: el método **visualizar( )** imprime un **String**, el método **pInt( )** imprime un **String** seguido de un **int,** y el médodo **pFlt( )** imprime un **String** seguido de un **float**. Por supuesto, en última instancia todos usan **System.out.println( )**.

Para generar números, el programa crea en primer lugar un objeto **Random**. Como no se le pasan parámetros en su creación, Java usa la hora actual como semilla para el generador de números aleatorio. El programa genera un conjunto de números aleatorios de distinto tipo con el objeto **Random** simplemente llamando a distintos métodos: **nextInt( )**, **nextLong( )**, **nextFloat( )** o **nextDouble( )**.

Cuando el operador módulo se usa con el resultado de un generador de números aleatorios, limita el resultado a un límite superior del operando menos uno (en este caso 99).

Los operadores unarios de suma y resta

El menos unario (-) y el más unario (+) son los mismos operadores que la resta y la suma binarios. El compilador averigua cuál de los dos usos es el pretendido por la manera de escribir la expresión. Por ejemplo, la sentencia:

```
x = -a;
```

tiene un significado obvio. El compilador es capaz de averiguar:

```
x = a * -b;
```

Pero puede que el lector llegue a confundirse, por lo que es más claro decir:

```
x = a * (-b);
```

El menos unario genera el valor negativo del valor dado. El más unario proporciona simetría con el menos unario, aunque no tiene ningún efecto.

Autoincremento y Autodecremento

Tanto Java, como C, está lleno de atajos. Éstos pueden simplificar considerablemente el tecleado del código, y aumentar o disminuir su legibilidad.

Dos de los atajos mejores son los operadores de incremento y decremento (que a menudo se llaman operadores de autoincremento y autodecremento). El operador de decremento es -- y significa "disminuir en una unidad". El operador de incremento es ++ y significa "incrementar en una unidad". Si **a** es un entero, por ejemplo, la expresión **++a** es equivalente a (**a** = **a** + **1**). Los operadores de incremento y decremento producen el valor de la variable como resultado.

Hay dos versiones de cada tipo de operador, llamadas, a menudo, versiones prefija y postfija. El preincremento quiere decir que el operador ++ aparece antes de la variable o expresión, y el postincremento significa que el operador ++ aparece después de la variable o expresión. De manera análoga, el predecremento quiere decir que el operador -- aparece antes de la variable o expresión, y el post-decremento significa que el operador -- aparece después de la variable o expresión. Para el preincremento y el predecremento (por ejemplo, **++a** o—**a**), la operación se lleva a cabo y se produce el valor. En el caso del postincremento y postdecremento (por ejemplo, **a++** o **a--**) se produce el valor y después se lleva a cabo la operación. Por ejemplo:

```
//: c03:AutoInc.java
// Mostrar el funcionamiento de los operadores ++ y --

public class AutoInc {
  public static void main (String[] args) {
    int i = 1;
    visualizar("i : " + i);
    visualizar("++i : " + ++i); // Pre-incremento
    visualizar("i++ : " + i++); // Post-incremento
```

```
    visualizar("i : " + i);
    visualizar("--i : " + --i); // Pre-decremento
    visualizar("i-- : " + i--); // Post-decremento
    visualizar("i : " + i);
  }
static void visualizar(String s) {
    System.out.println(s);
  }
} ///:~
```

La salida de este programa es:

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

Se puede pensar que con la forma prefija se consigue el valor después de que se ha hecho la operación, mientras que con la forma postfija se consigue el valor antes de que la operación se lleve a cabo. Éstos son los únicos operadores (además de los que implican asignación) que tienen efectos laterales. (Es decir, cambian el operando en vez de usarlo simplemente como valor.)

El operador de incremento es una explicación para el propio nombre del lenguaje C++, que significa "un paso después de C". En una de las primeras conferencias sobre Java, Bill Joy (uno de sus creadores), dijo que "Java=C++–" (C más más menos menos), tratando de sugerir que Java es C++ sin las partes duras no necesarias, y por consiguiente, un lenguaje bastante más sencillo. A medida que se progrese en este libro, se verá cómo muchas partes son más simples, y sin embargo, Java no es *mucho* más fácil que C++.

Operadores relacionales

Los operadores relacionales generan un resultado de tipo **boolean**. Evalúan la relación entre los valores de los operandos. Una expresión relacional produce **true** si la relación es verdadera, y **false** si la relación es falsa. Los operadores relacionales son menor que (<), mayor que (>), menor o igual que (<=), mayor o igual que (>=), igual que (==) y distinto que (!=). La igualdad y la desigualdad funcionan con todos los tipos de datos predefinidos, pero las otras comparaciones no funcionan con el tipo **boolean**.

Probando la equivalencia de objetos

Los operadores relacionales == y != funcionan con todos los objetos, pero su significado suele confundir al que programa en Java por primera vez. He aquí un ejemplo:

```
//: c03:Equivalencia.java

public class Equivalencia {
  public static void main(String[] args) {
```

```
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1 == n2);
    System.out.println(n1 != n2);
  }
} ///:~
```

La expresión **System.out.println(n1 == n2)** visualizará el resultado de la comparación de tipo lógico. Seguramente la salida debería ser **true** y después **false**, pues ambos objetos **Integer** son el mismo. Pero mientras que los *contenidos* de los objetos son los mismos, las referencias no son las mismas, y los operadores == y != comparan referencias a objetos. Por ello, la salida es, de hecho, **false** y después **true**. Naturalmente, esto sorprende a la gente al principio.

¿Qué ocurre si se desea comparar los contenidos de dos objetos? Es necesario utilizar el método especial **equals( )** que existe para todos los objetos (no tipos primitivos, que funcionan perfectamente con == y !=). He aquí cómo usarlo:

```
//: c03:MetodoComparacion.java

public class MetodoComparacion {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1.equals(n2));
  }
} ///:~
```

El resultado será **true**, tal y como se espera. Ah, pero no es así de simple. Si uno crea su propia clase, como ésta:

```
//:c03:MetodoComparacion2.java

class Valor {
  int i;
}

public class MetodoComparacion2 {
  public static void main(String[] args) {
    Valor v1 = new Valor();
    Valor v2 = new Valor();
    v1.i = v2.i = 100;
    System.out.println(v1.equals(v2));
  }
} ///:~
```

se obtiene como resultado **falso**. Esto se debe a que el comportamiento por defecto de **equals( )** es comparar referencias. Por tanto, a menos que se invalide **equals( )** en la nueva clase no se obtendrá el comportamiento deseado. Desgraciadamente no se mostrarán las invalidaciones hasta el Capítulo

7, pero debemos ser conscientes mientras tanto de la forma en que se comporta **equals( )** podría ahorrar algunos problemas.

La mayoría de clases de la biblioteca Java implementan **equals( )**, de forma que compara los contenidos de los objetos en vez de sus referencias.

Operadores lógicos

Los operadores lógicos AND (&&), OR (||) y NOT(!) producen un valor **lógico (true** o **false)** basado en la relación lógica de sus argumentos. Este ejemplo usa los operadores relacionales y lógicos:

```
//: c03:Logico.java
// Operadores relacionales y lógicos
import java.util.*;

public class Logico {
  public static void main(String[] args) {
    Random aleatorio = new Random();
    int i  = aleatorio.nextInt() % 100;
    int j = aleatorio.nextInt() % 100;
    visualizar("i = " + i);
    visualizar("j = " + j);
    visualizar("i > j es " + (i > j));
    visualizar("i < j es " + (i < j));
    visualizar("i >= j es " + (i >= j));
    visualizar("i <= j es " + (i <= j));
    visualizar("i == j es " + (i == j));
    visualizar("i != j es " + (i != j));

    // Tratar un int como un boolean no es legal en Java
  //! visualizar ("i && j es " + (i && j));
  //! visualizar ("i || j es " + (i || j));
  //! visualizar ("!i es " + !i);

    visualizar("(i<10) && (j<10) es " + ((i < 10) && (j < 10));
    visualizar("(i<10) || (j<10) es " + ((i < 10) || (j < 10));
  }
  static void visualizar(String s) {
    System.out.println(s);
  }
} ///:~
```

Sólo es posible aplicar AND, OR o NOT a valores **boolean**. No se puede construir una expresión lógica con valores que no sean de tipo **boolean**, cosa que sí se puede hacer en C y C++. Se pueden

ver intentos fallidos de hacer esto en las líneas que comienzan por //! en el ejemplo anterior. Sin embargo, las sentencias que vienen a continuación producen valores **lógicos** utilizando comparaciones relacionales, y después se usan operaciones lógicas en los resultados.

El listado de salida tendrá la siguiente apariencia:

```
i = 85;
j = 4;
i > j es true
i < j es false
i >= j es true
i <= j es false
i == j es false
i != es true
(i < 10) && (j < 10) es false
(i < 10) || (j > 10) es true
```

Obsérvese que un valor **lógico** se convierte automáticamente a formato de texto si se utiliza allí donde se espera un **String**.

Se puede reemplazar la definición **int** en el programa anterior por cualquier otro tipo de datos primitivo, excepto **boolean**. Hay que tener en cuenta, sin embargo, que la comparación de números en coma flotante es muy estricta. Un número que sea diferente por muy poco de otro número sigue siendo "no igual". Un número infinitamente próximo a cero es distinto de cero.

Cortocircuitos

Al manipular los operadores lógicos se puede entrar en un fenómeno de "cortocircuito". Esto significa que la expresión se evaluará únicamente *hasta* que se pueda determinar sin ambigüedad la certeza o falsedad de toda la expresión. Como resultado, podría ocurrir que no sea necesario evaluar todas las partes de la expresión lógica. He aquí un ejemplo que muestra el funcionamiento de los cortocircuitos:

```
//: c03:CortoCircuito.java
// Demuestra el comportamiento de los cortocircuitos con operadores
lógicos.

public class CortoCircuito {
  static boolean prueba1(int val) {
    System.out.println("prueba1(" + val + ")");
    System.out.println("resultado: " + (val < 1));
    return val < 1;
  }
  static boolean prueba2(int val) {
    System.out.println("prueba2(" + val + ")");
    System.out.println("resultado: " + (val < 2));
    return val < 2;
  }
```

```
  static boolean prueba3(int val) {
    System.out.println("prueba3(" + val + ")");
    System.out.println("resultado: " + (val < 3));
    return val < 3;
  }
  public static void main(String[] args) {
    if(prueba1(0) && prueba2(2) && prueba3(2))
      System.out.println("La expresión es verdadera");
    else
      System.out.println("La expresión es falsa");
  }
} ///:~
```

Cada test lleva a cabo una comparación con el argumento pasado y devuelve verdadero o falso. También imprime información para mostrar lo que se está invocando. Las comprobaciones se usan en la expresión:

```
if (prueba1(0) && prueba2(2) && prueba3(2))
```

Naturalmente uno podría pensar que se ejecutarían las tres pruebas, pero en la salida se muestra de otra forma:

```
prueba1(0)
resultado: true
prueba2(2)
resultado: false
la expresión es falsa
```

La primera prueba produjo un resultado **verdadero**, de forma que la evaluación de la expresión continúa. Sin embargo, el segundo test produjo un resultado **falso**. Puesto que esto significa que toda la expresión va a ser **falso** ¿por qué continuar evaluando el resto de la expresión? Podría ser costoso. Ésa es precisamente la razón para realizar un cortocircuito; es posible lograr un incremento potencial de rendimiento si no es necesario evaluar todas las partes de la expresión lógica.

Operadores de bit

Los operadores a nivel de bit permiten manipular bits individuales de la misma forma que si fueran tipos de datos primitivos íntegros. Los operadores de bit llevan a cabo álgebra lógica con los bits correspondientes de los dos argumentos, para producir el resultado.

Los operadores a nivel de bit provienen de la orientación a bajo nivel de C, para la manipulación directa del hardware y el establecimiento de los bits de los registros de hardware. Java se diseñó originalmente para ser empotrado en las cajas *set-top* de los televisores, de forma que esta orientación de bajo nivel tenía sentido. Sin embargo, probablemente no se haga mucho uso de estos operadores de nivel de bit.

El operador de bit AND (**&**) produce un uno a la salida si los dos bits de entrada son unos; si no, produce un cero. El operador de bit OR (**|**) produce un uno en la salida si cualquiera de los bits de

entrada es un uno, y produce un cero sólo si los dos bits de entrada son cero. El operador de bit OR EXCLUSIVO o XOR (^), produce un uno en la salida si uno de los bits de entrada es un uno, pero no ambos. El operador de bit NOT (~, también llamado operador de *complemento a uno*) es un operador unario; toma sólo un argumento. (Todos los demás operadores de bits son operadores binarios.) El operador de bit NOT produce el contrario del bit de entrada —un uno si el bit de entrada es cero y un cero si el bit de entrada es un uno.

Los operadores de bit y lógicos utilizan los mismos caracteres, por lo que ayuda tener algún mecanismo mnemónico para ayudar a recordar su significado: dado que los bits son "pequeños", sólo hay un carácter en los operadores de bits.

Los operadores de bit se pueden combinar con el signo = para unir la operación a una asignación: **&=**, |= y ^= son válidos (dado que ~ es un operador unario, no puede combinarse con el signo =).

El tipo **boolean** se trata como un valor de un bit, por lo que es en cierta medida distinto. Se puede llevar a cabo un AND, OR o XOR de bit, pero no se puede realizar un NOT de bit (se supone que para evitar la confusión con el NOT lógico). Para los datos de tipo **boolean**, los operadores de bit tienen el mismo efecto que los operadores lógicos, excepto en que no tienen capacidad de hacer cortocircuitos. Además, los operadores de bit sobre datos de tipo **boolean** incluyen un operador XOR lógico no incluido bajo la lista de operadores "lógicos". Hay que tratar de evitar los datos de tipo **boolean** en las expresiones de desplazamiento, descritas a continuación.

Operadores de desplazamiento

Los operadores de desplazamiento también manipulan bits. Sólo se pueden utilizar con tipos primitivos enteros. El operador de desplazamiento a la izquierda (<<) provoca que el operando de la izquierda del operador sea desplazado a la izquierda, tantos bits como se especifique tras el operador (insertando ceros en los bits menos significativos). El operador de desplazamiento a la derecha con signo (>>) provoca que el operando de la izquierda del operador sea desplazado a la derecha el número de bits que se especifique tras el operador. El desplazamiento a la derecha con signo >> utiliza la *extensión de signo*: si el valor es positivo se insertan ceros en los bits más significativos; si el valor es negativo, se insertan unos en los bits más significativos. Java también ha incorporado el operador de rotación a la derecha sin signo >>>, que utiliza la *extensión cero*: independientemente del signo, se insertan ceros en los bits más significativos. Este operador no existe ni en C ni en C++.

Si se trata de desplazar un **char**, un **byte** o un **short**, éste será convertido a **int** antes de que el desplazamiento tenga lugar y el resultado será también un **int**. Sólo se utilizarán los cinco bits menos significativos de la parte derecha. Esto evita que se desplace un número de bits mayor al número de bits de un **int**. Si se está trabajando con un **long**, se logrará un resultado **long**. Sólo se usarán los seis bits menos significativos de la parte derecha, por lo que no es posible desplazar más bits que los que hay en un **long**.

Los desplazamientos pueden combinarse con el signo igual (<<= o >>= o >>>=). El ivalor se reemplaza por el ivalor desplazado por el dvalor. Hay un problema, sin embargo, con el desplazamiento sin signo a la derecha combinado con la asignación. Si se utiliza con un **byte** o **short** no se logra el resultado correcto. En vez de esto, los datos son convertidos a **int** y desplazados a la derecha, y

teriormente se truncan al ser asignados de nuevo a sus variables, por lo que en esos casos el resultado suele ser **-1**. El ejemplo siguiente demuestra esto:

```
//: c03:DesplDatosSinSigno.java
// Prueba del desplazamiento a la derecha sin signo.

public class DesplDatosSinSigno {
  public static void main(String[] args) {
    int i = -1;
    i >>>= 10;
    System.out.println(i);
    long l = -1;
    l >>>= 10;
    System.out.println(l);
    short s = -1;
    s >>>= 10;
    System.out.println(s);
    byte b = -1;
    b >>>= 10;
    System.out.println(b);
    b = -1;
    System.out.println(b>>>10);
  }
} ///:~
```

En la última línea, no se asigna el valor resultante de nuevo a **b**, sino que se imprime directamente para que se dé el comportamiento correcto.

He aquí un ejemplo que demuestra el uso de todos los operadores que involucran a bits:

```
//: c03:ManipulacionBits.java
// Utilizando los operadores de bit.
import java.util.*;

public class ManipulacionBits {
  public static void main(String[] args) {
    Random aleatorio = new Random();
    int i = aleatorio.nextInt();
    int j = aleatorio.nextInt();
    pBinInt("-1", -1);
    pBinInt("+1", +1);
    int posmax = 2147483647;
    pBinInt("posmax", posmax);
    int negmax = -2147483648;
    pBinInt("negmax", negmax);
    pBinInt("i", i);
    pBinInt("~i", ~i);
```

```
    pBinInt("-i", -i);
    pBinInt("j", j);
    pBinInt("i & j", i & j);
    pBinInt("i | j", i | j);
    pBinInt("i ^ j", i ^ j);
    pBinInt("i << 5", i << 5);
    pBinInt("i >> 5", i >> 5);
    pBinInt("(~i) >> 5", (~i) >> 5);
    pBinInt("i >>> 5", i >>> 5);
    pBinInt("(~i) >>> 5", (~i) >>> 5);

    long l = aleatorio.nextLong();
    long m = aleatorio.nextLong();
    pBinLong("-1L", -1L);
    pBinLong("+1L", +1L);
    long ll = 9223372036854775807L;
    pBinLong("maxpos", ll);
    long lln = -9223372036854775808L;
    pBinLong("maxneg", lln);
    pBinLong("l", l);
    pBinLong("~l", ~l);
    pBinLong("-l", -l);
    pBinLong("m", m);
    pBinLong("l & m", l & m);
    pBinLong("l | m", l | m);
    pBinLong("l ^ m", l ^ m);
    pBinLong("l << 5", l << 5);
    pBinLong("l >> 5", l >> 5);
    pBinLong("(~l) >> 5", (~l) >> 5);
    pBinLong("l >>> 5", l >>> 5);
    pBinLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void pBinInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binario: ");
    System.out.print("    ");
    for(int j = 31; j >=0; j--)
    if(((1 << j) &  i) != 0)
      System.out.print("1");
    else
      System.out.print("0");
    System.out.println();
  }
  static void pBinLong(String s, long l) {
    System.out.println(
```

```
      s + ", long: " + l + ", binario: ");
    System.out.print("    ");
    for(int i = 63; i >=0; i--)
      if(((1L << i) & l) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
} ///:~
```

Los dos métodos del final, **pBinInt( )** y **pBinLong( )** toman un **int** o un **long**, respectivamente, y lo imprimen en formato binario junto con una cadena de caracteres descriptiva. De momento, ignoraremos la implementación de estos métodos.

Se habrá dado cuenta el lector del uso de **System.out.print( )** en vez de **System.out.println( )**. El método **print( )** no finaliza con un salto de línea, por lo que permite ir visualizando una línea por fragmentos.

Además de demostrar el efecto de todos los operadores de bit para **int** y **long**, este ejemplo también muestra los valores mínimo, el máximo, +1 y –1 para **int** y para **long**, por lo que puede verse qué aspecto tienen. Nótese que el bit más significativo representa el signo: 0 significa positivo, y 1 significa negativo. La salida de la porción **int** tiene la apariencia siguiente:

```
-1, int: -1, binario:
  11111111111111111111111111111111
+1, int: 1, binario:
  00000000000000000000000000000001
posmax, int: 2147483647, binario:
  01111111111111111111111111111111
negmax, int: -2147483648, binario:
  10000000000000000000000000000000
i, int: 59081716, binario:
  00000011100001011000001111110100
~i, int: -59081717, binario:
  11111100011110100111110000001011
-i, int: -59081716, binarios:
  11111100011110100111110000001100
j, int: 198850956, binario:
  00001011110110100011100110001100
i & j, int: 58720644, binario:
  00000011100000000000000110000100
i | j, int: 199212028, binario:
  00001011110111111011101111111100
i ^ j, int: 140491384, binario:
  00001000010111111011101001111000
```

```
i << 5, int: 1890614912, binario:
  01110000101100000111111010000000
i >> 5, int: 1846303, binario:
  00000000000111000010110000011111
(~ i) >>5, int: -1846304, binario:
  11111111111000111101001111100000
i >>> 5, int: 1846303, binario:
  00000000000111000010110000011111
(~ i) >>> 5, int: 132371424, binario:
  00000111111000111101001111100000
```

La representación binaria de los números se denomina también *complemento a dos con signo.*

Operador ternario if-else

Este operador es inusual por tener tres operandos. Verdaderamente es un operador porque produce un valor, a diferencia de la sentencia if-else ordinaria que se verá en la siguiente sección de este capítulo. La expresión es de la forma:

```
exp-booleana ? valor0 : valor1
```

Si el resultado de la evaluación exp-boolean es **true**, se evalúa *valor0* y su resultado se convierte en el valor producido por el operador. Si exp-booleana es **false**, se evalúa *valor1* y su resultado se convierte en el valor producido por el operador.

Por supuesto, podría usarse una sentencia **if-else** ordinaria (descrita más adelante), pero el operador ternario es mucho más breve. Aunque C (del que es originario este operador) se enorgullece de ser un lenguaje sencillo, y podría haberse introducido el operador ternario en parte por eficiencia, deberíamos ser cautelosos a la hora de usarlo cotidianamente —es fácil producir código ilegible.

El operador condicional puede usarse por sus efectos laterales o por el valor que produce, pero en general se desea el valor, puesto que es éste el que hace al operador distinto del **if-else**. He aquí un ejemplo:

```
static int ternario(int i) {
  return i < 10 ? i * 100 : i * 10;
}
```

Este código, como puede observarse, es más compacto que el necesario para escribirlo sin el operador ternario:

```
static int alternativo(int i) {
  if (i < 10)
    return i * 100;
  else
    return i * 10;
}
```

La segunda forma es más sencilla de entender, y no requiere de muchas más pulsaciones. Por tanto, hay que asegurarse de evaluar las razones a la hora de elegir el operador ternario.

El operador coma

La coma se usa en C y C++ no sólo como un separador en las listas de parámetros a funciones, sino también como operador para evaluación secuencial. El único lugar en que se usa el *operador* coma en Java es en los bucles **for**, que serán descritos más adelante en este capítulo.

El operador de String +

Hay un uso especial en Java de un operador: el operador + puede utilizarse para concatenar cadenas de caracteres, como ya se ha visto. Parece un uso natural del + incluso aunque no encaje con la manera tradicional de usar el +. Esta capacidad parecía una buena idea en C++, por lo que se añadió la *sobrecarga de operadores* a C++, para permitir al programador de C++ añadir significados a casi todos los operadores. Por desgracia, la sobrecarga de operadores combinada con algunas otras restricciones de C++, parece convertirse en un aspecto bastante complicado para que los programadores la usen al diseñar sus clases. Aunque la sobrecarga de operadores habría sido mucho más fácil de implementar en Java que en C++, se seguía considerando que se trataba de un aspecto demasiado complicado, por lo que los programadores de Java no pueden implementar sus propios operadores sobrecargados como pueden hacer los programadores de C++.

El uso del + de **String** tiene algún comportamiento interesante. Si una expresión comienza con un **String**, entonces todos los operandos que le sigan deben ser de tipo **String** (recuerde que el compilador convertirá una secuencia de caracteres entre comas en un **String**):

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

Aquí, el compilador Java convertirá a **x**, **y** y **z** en sus representaciones **String** en vez de sumarlas. Mientras que si se escribe:

```
System.out.printl(x + sString);
```

Java convertirá **x** en un **String**.

Pequeños fallos frecuentes al usar operadores

Uno de los errores frecuentes al utilizar operadores es intentar no utilizar paréntesis cuando se tiene la más mínima duda sobre cómo se evaluará una expresión. Esto sigue ocurriendo también en Java.

Un error extremadamente frecuente en C y C++ es éste:

```
while (x = y) {
  // ...
}
```

El programador estaba intentando probar una equivalencia (==) en vez de hacer una asignación. En C y C++ el resultado de esta asignación siempre será **true** si **y** es distinta de cero, y probablemente se entrará en un bucle infinito. En Java, el resultado de esta expresión no es un **boolean**, y el compilador espera un **boolean** pero no convertirá el **int** en **boolean**, por lo que dará el conveniente error en tiempo de compilación, y capturará el problema antes de que se intente siquiera ejecutar el programa. De esta forma, esta trampa jamás puede ocurrir en Java. (El único momento en que no se obtendrá un error en tiempo de compilación es cuando **x** e **y** sean **boolean**, en cuyo caso **x = y** es una expresión legal, y en el caso anterior, probablemente un error.)

Un problema similar en C y C++ es utilizar los operadores de bit AND y OR, en vez de sus versiones lógicas. Los AND y OR de bit utilizan uno de los caracteres (**&** o **|**) y los AND y OR lógicos utilizan dos (**&&** y **||**). Como ocurre con el = y el ==, es fácil escribir sólo uno de los caracteres en vez de ambos. En Java, el compilador vuelve a evitar esto porque no los permite utilizar con operadores incorrectos.

Operadores de conversión

La palabra *conversión* se utiliza con el sentido de "convertir[1] a un molde". Java convertirá automáticamente un tipo de datos en otro cuando sea adecuado. Por ejemplo, si se asigna un valor entero a una variable de coma flotante, el compilador convertirá automáticamente el **int** en **float**. La conversión permite llevar a cabo estas conversiones de tipos de forma explícita, o forzarlas cuando no se diesen por defecto.

Para llevar a cabo una conversión, se pone el tipo de datos deseado (incluidos todos los modificadores) entre paréntesis a la izquierda de cualquier valor. He aquí un ejemplo:

```
void conversiones() {
  int i = 200;
  long l = (long)i;
  long l2 = (long)200;
}
```

Como puede verse, es posible llevar a cabo una conversión, tanto con un valor numérico, como con una variable. En las dos conversiones mostradas, la conversión es innecesaria, dado que el compilador convertirá un valor **int** en **long** cuando sea necesario. No obstante, se permite usar conversiones innecesarias para hacer el código más limpio. En otras situaciones, puede ser esencial una conversión para lograr que el código compile.

En C y C++, las conversiones pueden conllevar quebraderos de cabeza. En Java, la conversión de tipos es segura, con la excepción de que al llevar a cabo una de las denominadas *conversiones reductoras* (es decir, cuando se va de un tipo de datos que puede mantener más información a otro que no puede contener tanta) se corre el riesgo de perder información. En estos casos, el compilador fuerza a hacer una conversión explícita, diciendo, de hecho, "esto puede ser algo peligroso de hacer

[1] N. del Traductor: Casting se traduce aquí por convertir.

—si quieres que lo haga de todas formas, tiene que hacer la conversión de forma explícita". Con una *conversión extensora* no es necesaria una conversión explícita porque el nuevo tipo es capaz de albergar la información del viejo tipo sin que se pierda nunca ningún bit.

Java permite convertir cualquier tipo primitivo en cualquier otro tipo, excepto **boolean**, que no permite ninguna conversión. Los tipos clase no permiten ninguna conversión. Para convertir una a otra debe utilizar métodos especiales (**String** es un caso especial y se verá más adelante en este libro que los objetos pueden convertirse en una *familia* de tipos; un **Roble** puede convertirse en **Árbol** y viceversa, pero esto no puede hacerse con un tipo foráneo como **Roca**.)

Literales

Generalmente al insertar un valor literal en un programa, el compilador sabe exactamente de qué tipo hacerlo. Sin embargo, en ocasiones, el tipo es ambiguo. Cuando ocurre esto es necesario guiar al compilador añadiendo alguna información extra en forma de caracteres asociados con el valor literal. El código siguiente muestra estos caracteres:

```
//: c03:Literales.java

class Literales {
  char c = 0xffff; // Carácter máximo valor hexadecimal
  byte b = 0x7f; // Máximo byte valor hexadecimal
  short s = 0x7fff; // Máximo short valor hexadecimal
  int i1 = 0x2f; // Hexadecimal (minúsculas)
  int i2 = 0X2F; // Hexadecimal (mayúsculas)
  int i3 = 0177; // Octal (Cero delantero)
  // Hex y Oct también funcionan con long.
  long n1 = 200L; // sufijo long
  long n2 = 200l; // sufijo long
  long n3 = 200;
  //! long l6(200); // prohibido
  float f1 = 1;
  float f2 = 1F; // sufijo float
  float f3 = 1f; // sufijo float
  float f4 = 1e-45f; // 10 elevado a -45
  float f5 = 1e+9f; // sufijo float
  double d1 = 1d; // sufijo double
  double d2 = 1D; // sufijo double
  double d3 = 47e47d; // 10 elevado a 47
  } ///:~
```

La base 16 (hexadecimal), que funciona con todos los tipos de datos enteros, se representa mediante un **0x** o **0X** delanteros, seguidos de 0–9 y a–f, tanto en mayúsculas como en minúsculas. Si se trata de inicializar una variable con un valor mayor que el que puede albergar (independientemente de la forma numérica del valor), el compilador emitirá un mensaje de error. Fíjese en el código anterior, los valores hexadecimales máximos posibles para **char**, **byte** y **short**. Si se excede de éstos, el compi-

lador generará un valor **int** automáticamente e informará de la necesidad de hacer una conversión reductora para llevar a cabo la asignación. Se sabrá que se ha traspasado la línea.

La base 8 (octal) se indica mediante un cero delantero en el número, y dígitos de 0 a 7. No hay representación literal de números binarios en C, C++ o Java.

El tipo de un valor literal lo establece un carácter arrastrado por éste. Sea en mayúsculas o minúsculas, **L** significa **long**, **F** significa **float**, y **D** significa **double**.

Los exponentes usan una notación que yo a veces encuentro bastante desconcertante: **1,39 e-47f**. En ciencias e ingeniería, la "e" se refiere a la base de los logaritmos naturales, aproximadamente 2,718. (Hay un valor **double** mucho más preciso en Java, denominado **Math.E**.) Éste se usa en expresión exponencial, como 1,39 e^{-47}, que quiere decir 1,39 x $2{,}718^{-47}$. Sin embargo, cuando se inventó *Fortran* se decidió que la **e** querría indicar "diez elevado a la potencia" lo cual es una mala decisión, pues *Fortran* fue diseñado para ciencias e ingeniería y podría pensarse que los diseñadores deben ser conscientes de que se ha introducido semejante ambigüedad[2]. En cualquier caso, esta costumbre siguió en C y C++, y ahora en Java. Por tanto, si uno está habituado a pensar que **e** es la base de los logaritmos naturales, tendrá que hacer una traslación mental al ver una expresión como **1,39** $\mathbf{e^{-47f}}$ en Java; significa **1,39 *** $\mathbf{10^{-47}}$.

Nótese que no es necesario utilizar el carácter final cuando el compilador puede averiguar el tipo apropiado. Con

```
long n3 = 200;
```

no hay ambigüedad, por lo que una **L** tras el 200 sería superflua. Sin embargo, con

```
float f4 = 1e-47f; // 10 elevado a
```

el compilador, normalmente, tomará los números exponenciales como *double*, de forma que sin la **f** arrastrada dará un error indicando que es necesario hacer una conversión de **double** en un **float**.

Promoción

Al hacer operaciones matemáticas o de bit sobre tipos de datos primitivos, se descubrirá que si son más pequeños que un **int** (es decir, **char**, **byte**, o **short**), estos valores se promocionarán a **int** antes de hacer las operaciones, y el valor resultante será de tipo **int**. Por tanto, si se desea asignar el valor devuelto, de nuevo al tipo de menor tamaño, será necesario utilizar una conversión. (Y dado

---

[2] John Kirkham escribe: "Empecé a trabajar con computadores en 1962 utilizando FORTRAN II en un IBM 1620. En ese tiempo, y a través de los años sesenta y setenta, FORTRAN era un lenguaje todo en mayúsculas. Esto empezó probablemente porque muchos de los primeros dispositivos de entrada eran viejas unidades de teletipo que utilizaban código Baudot de 5 bits, que no tenía capacidad de empleo de minúsculas. La 'E' para la notación exponencial era también siempre mayúscula y nunca se confundía con la base de los logaritmos naturales 'e', que siempre era minúscula. La 'E' simplemente quería decir siempre exponencial, que era la base del sistema de numeración utilizado —generalmente 10. En ese momento se comenzó a extender entre los programadores el sistema octal. Aunque yo nunca lo vi usar, si hubiera visto un número octal en notación exponencial, habría considerado que tenía base 8. La primera vez que recuerdo ver un exponencial utilizando una 'e' minúscula fue al final de los setenta, y lo encontré bastante confuso. El problema aumentó cuando la 'e' se introdujo en FORTRAN, a diferencia de sus principios. De hecho, nosotros teníamos funciones para usar cuando realmente se quería usar la base logarítmica natural, pero todas ellas eran en mayúsculas".

que se está haciendo una asignación, de nuevo hacia un tipo más pequeño, se podría estar perdiendo información.) En general, el tipo de datos de mayor tamaño en una expresión será el que determine el tamaño del resultado de esa expresión; si se multiplica un **float** y un **double**, el resultado será **double**; si se suman un **int** y un **long**, el resultado será **long**.

Java no tiene "sizeof"

En C y C++, el operador **sizeof( )** satisface una necesidad específica: nos dice el número de bits asignados a elementos de datos. La necesidad más apremiante de **sizeof( )** en C y C++ es la portabilidad. Distintos tipos de datos podrían tener distintos tamaños en distintas máquinas, por lo que el programador debe averiguar cómo de grandes son estos tipos de datos, al llevar a cabo operaciones sensibles al tamaño. Por ejemplo, un computador podría almacenar enteros en 32 bits, mientras que otro podría almacenar enteros como 16 bits. Los programas podrían almacenar enteros con valores más grandes en la primera de las máquinas. Como podría imaginarse, la portabilidad es un gran quebradero de cabeza para los programadores de C y C++.

Java no necesita un operador **sizeof( )** para este propósito porque todos los tipos de datos tienen los mismos tamaños en todas las máquinas. No es necesario pensar en la portabilidad a este nivel —está intrínsecamente diseñada en el propio lenguaje.

Volver a hablar acerca de la precedencia

Tras oír quejas en uno de mis seminarios, relativas a la complejidad de recordar la precedencia de los operadores uno de mis alumnos sugirió un recurso mnemónico que es simultáneamente un comentario (en inglés): "Ulcer Addicts Really Like C A lot."

<table>
<tr><th>Mnemónico</th><th>Tipo de operador</th><th>Operador</th></tr>
<tr><td>Ulcer</td><td>Unario</td><td>+ – ++ –</td></tr>
<tr><td>Addicts</td><td>Aritméticos (y de desplazamiento)</td><td>* / % + – << >></td></tr>
<tr><td>Really</td><td>Relacional</td><td>> < >= <= == !=</td></tr>
<tr><td>Like</td><td>Lógicos (y de bit)</td><td>&& || & | ^</td></tr>
<tr><td>C</td><td>Condicional (ternario)</td><td>A > B ? X : Y</td></tr>
<tr><td>A Lot</td><td>Asignación</td><td>= (y asignaciones compuestas como *=)</td></tr>
</table>

Por supuesto, con los operadores de desplazamiento y de bit distribuidos por toda la tabla, el recurso mnemónico no es perfecto, pero funciona para las operacione de no bit.

Un compendio de operadores

El ejemplo siguiente muestra qué tipos de datos primitivos pueden usarse como operadores particulares. Básicamente, es el mismo ejemplo repetido una y otra vez, pero usando distintos tipos de datos primitivos. El fichero se compilará sin error porque las líneas que causarían errores están marcadas como comentarios con un //!.

```
//: c03:TodosOperadores.java
// Prueba todos los operadores con
// todos los tipos de datos para probar
// cuáles son aprobados por el compilador de Java.

class TodosOperadores {
  // Para aceptar los resultados de un test booleano:
  void f(boolean b) {}
  void pruebaBool(boolean x, boolean y) {
    // Operadores aritméticos:
    //! x = x * y;
    //! x = x / y;
    //! x = x % y;
    //! x = x + y;
    //! x = x - y;
    //! x++;
    //! x--;
    //! x = +y;
    //! x = -y;
    // Relacionales y lógicos:
    //! f(x > y);
    //! f(x >= y);
    //! f(x < y);
    //! f(x <= y);
    f(x == y);
    f(x != y);
    f(!y);
    x = x && y;
    x = x || y;
    // Operadores de bit:
    //! x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Asignación compuesta:
```

```
    //! x += y;
    //! x -= y;
    //! x *= y;
    //! x /= y;
    //! x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Conversión:
    //! char c = (char)x;
    //! byte B = (byte)x;
    //! short s = (short)x;
    //! int i = (int)x;
    //! long l = (long)x;
    //! float f = (float)x;
    //! double d = (double)x;
  }
  void pruebaChar(char x, char y) {
    // Operadores aritméticos:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char)+y;
    x = (char)-y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores de bit:
    x= (char)~y;
    x = (char)(x & y);
    x = (char)(x | y);
```

```
    x = (char)(x ^ y);
    x = (char)(x << 1);
    x = (char)(x >> 1);
    x = (char)(x >>> 1);
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Conversión:
    //! boolean b = (boolean)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void pruebaByte(byte x, byte y) {
    // Operadores aritméticos:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte)+ y;
    x = (byte)- y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
```

```
    //! f(x || y);
    // Operadores de bit:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
    x = (byte)(x >> 1);
    x = (byte)(x >>> 1);
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Conversión:
    //! boolean b = (boolean)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void pruebaShort(short x, short y) {
    // Operadores aritméticos:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
```

```
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores de bit:
    x = (short)~y;
    x = (short)(x & y);
    x = (short)(x | y);
    x = (short)(x ^ y);
    x = (short)(x << 1);
    x = (short)(x >> 1);
    x = (short)(x >>> 1);
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Conversión:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void pruebaInt(int x, int y) {
    // Operadores aritméticos:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
```

```
    x = -y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores de bit:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Conversión:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void pruebaLong(long x, long y) {
    // Operadores aritméticos:
    x = x * y;
    x = x / y;
    x = x % y;
```

```
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores de bit:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Conversión:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
  }
```

```
void pruebaFloat(float x, float y) {
  // Operadores aritméticos:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Relacionales y lógicos:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Operadores de bit:
  //! x = ~y;
  //! x = x & y;
  //! x = x | y;
  //! x = x ^ y;
  //! x = x << 1;
  //! x = x >> 1;
  //! x = x >>> 1;
  // Asignación compuesta:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  //! x <<= 1;
  //! x >>= 1;
  //! x >>>= 1;
  //! x &= y;
  //! x ^= y;
  //! x |= y;
  // Conversión:
  //! boolean b = (boolean)x;
  char c = (char)x;
  byte B = (byte)x;
```

```
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
  }
  void pruebaDouble(double x, double y) {
    // Operadores aritméticos:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores de bit:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
```

```
    //! x |= y;
    // Conversión:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
  }
} ///:~
```

Fíjese que **boolean** es bastante limitado. Se le pueden asignar los valores **true** y **false**, y se puede comprobar su validez o falsedad, pero no se pueden sumar valores lógicos o llevar a cabo ningún otro tipo de operación sobre ellos.

En **char**, **byte** y **short** se puede ver el efecto de promoción con los operadores aritméticos. Cada operación aritmética que se haga con estos tipos genera como resultado un **int**, que debe ser explícitamente convertido para volver al tipo original (una conversión reductora que podría implicar pérdida de información) para volver a ser asignado a ese tipo. Con los valores **int**, sin embargo, no es necesaria ninguna conversión, porque todo es ya un **int**. Aunque no hay que relajarse pensando que todo está ya a salvo. Si se multiplican dos **valores de tipo int** lo suficientemente grandes, se desbordará el resultado. Esto se demuestra en el siguiente ejemplo:

```
//: c03:Desbordamiento.java
// ¡Sorpresa! Java permite desbordamientos.
public class Desbordamiento {
  public static void main(String[] args) {
    int grande = 0x7fffffff; // Valor entero máximo
    visualizar("grande = " + grande);
    int mayor = grande * 4;
    visualizar("mayor = " + mayor);
  }
  static void visualizar(String s) {
    System.out.println(s);
  }
} ///:~
```

La salida de esto es:

```
grande = 2147483647
mayor = -4
```

y no se recibe ningún error ni advertencia proveniente del compilador, ni excepciones en tiempo de ejecución. Java es bueno, pero no tanto.

La asignaciones compuestas no requieren conversiones para **char**, **byte** o **short**, incluso aunque estén llevando a cabo promociones que tienen los mismos resultados que los operadores aritméticos directos. Por otro lado, la falta de conversión, definitivamente, simplifica el código.

Se puede ver que, con la excepción de **boolean**, cualquier tipo primitivo puede convertirse a otro tipo primitivo. De nuevo, debemos ser conscientes del efecto de la conversión reductora cuando se hace una conversión a un tipo menor. Si no, se podría perder información sin saberlo durante la conversión.

Control de ejecución

Java utiliza todas las sentencias de control de ejecución de C, de forma que si se ha programado con C o C++, la mayoría de lo que se ha visto será familiar. La mayoría de los lenguajes procedurales tienen algún tipo de sentencia de control, y casi siempre hay solapamiento entre lenguajes. En Java, las palabras clave incluyen **if-else**, **while**, **do-while**, **for**, y una sentencia de selección denominada **switch**. Java, sin embargo, no soporta el siempre perjudicial **goto** (lo que podría seguir siendo la manera más expeditiva de solventar cierto tipo de problemas). Todavía se puede hacer un salto del estilo del "goto", pero es mucho más limitado que un **goto** típico.

True y false

Todas las sentencias condicionales utilizan la certeza o falsedad de una expresión de condición para determinar el cauce de ejecución. Un ejemplo de una expresión condicional es **A == B**. Ésta hace uso del operador condicional == para ver si el valor de **A** es equivalente al valor de **B**. La expresión devuelve **true** o **false**. Cualquiera de los operadores relacionales vistos anteriormente en este capítulo puede usarse para producir una sentencia condicional. Fíjese que Java no permite utilizar un número como un **boolean**, incluso aunque está permitido en C y C++ (donde todo lo distinto de cero es verdadero, y cero es falso). Si se quiere usar un valor que no sea lógico en una conducción lógica, como **if(a)**, primero es necesario convertirlo a un valor **boolean** utilizando una expresión condicional, como **if(a!=0)**.

If-else

La sentencia **if-else** es probablemente la manera más básica de controlar el flujo de un programa. El **else** es opcional, por lo que puede usarse **if** de dos formas:

```
if(expresión condicional)
  sentencia
```

o

```
if(expresión condicional)
  sentencia
else
  sentencia
```

La expresión condicional debe producir un resultado **boolean**. La *sentencia* equivale bien a una sentencia simple acabada en un punto y coma, o a una sentencia compuesta, que es un conjunto de sentencias simples encerradas entre llaves. Cada vez que se use la palabra *sentencia*, siempre implicará que ésta puede ser simple o compuesta.

He aquí un método **prueba( )** como ejemplo de **if-else**. Se trata de un método que indica si un número dicho en un acertijo es mayor, menor o equivalente al número solución:

```
//: c03:IfElse.java
public class IfElse {
  static int prueba(int intento, int solucion) {
  int resultado = 0;
  if(intento > solucion)
     resultado = +1;
  else if(intento < solucion)
     resultado = -1;
  else
     resultado = 0; // Coincidir
  return resultado;
  }
  public static void main(String[] args) {
  System.out.println(prueba(10, 5));
  System.out.println(prueba(5, 10));
  System.out.println(prueba(5, 5));
  }
} ///:~
```

Es frecuente alinear el cuerpo de una sentencia de control de flujo, de forma que el lector pueda determinar fácilmente dónde empieza y dónde acaba.

return

La palabra clave **return** tiene dos propósitos: especifica qué valor devolverá un método (si no tiene un valor de retorno **void**), y hace que el valor se devuelva inmediatamente. El método **prueba( )** puede reescribirse para sacar ventaja de esto:

```
//: c03:IfElse2.java
public class IfElse2 {
  static int prueba(int intento, int solucionar) {
  int resultado = 0;
  if(intento > solucionar)
  return +1;
  else if(intento < solucionar)
return -1;
else
  return 0; // Coincidir
```

```
    }
    public static void main(String[] args) {
    System.out.println(prueba(10, 5));
    System.out.println(prueba(5, 10));
    System.out.println(prueba(5, 5));
    }
} ///:~
```

No hay necesidad de **else** porque el método no continuará ejecutándose una vez que se ejecute el **return**.

Iteración

Las sentencias **while**, **do-while** y **for** son para el control de bucles, y en ocasiones se clasifican como *sentencias de iteración*. Se repite una *sentencia* hasta que la expresión *Condicional* controladora se evalúe a falsa. La forma de un bucle **while** es:

```
while(Expresión-Condicional)
sentencia
```

La *expresión condicional* se evalúa al comienzo de cada interación del bucle, y de nuevo antes de cada iteración subsiguiente de la *sentencia*.

He aquí un ejemplo sencillo que genera números aleatorios hasta que se dé una condición determinada:

```
//: c03:PruebaWhile.java
// Muestra el funcionamiento del bucle while.

public class PruebaWhile {
  public static void main(String[] args) {
  double r = 0;
  while(r < 0.99d) {
    r = Math.random();
    System.out.println(r);
     }
   }
} ///:~
```

Este ejemplo usa el método **estático random( )** de la biblioteca **Math**, que genera un valor **double** entre 0 y 1. (Incluye el 0, pero no el 1.) La expresión condicional para el **while** dice "siga haciendo este bucle hasta que el número sea 0,99 o mayor". Cada vez que se ejecute este programa, se logrará un listado de números de distinto tamaño.

do-while

La forma del **do-while** es

```
do
  sentencia
while (Expresión condicional);
```

La única diferencia entre **while** y **do-while** es que la sentencia del **do-while** se ejecuta siempre, al menos, una vez, incluso aunque la expresión se evalúe como falsa la primera vez. En un **while**, si la condicional es falsa la primera vez, la sentencia no se ejecuta nunca. En la práctica, **do-while** es menos común que **while**.

for

Un bucle **for** lleva a cabo la inicialización antes de la primera iteración. Después, lleva a cabo la comprobación condicional y, al final de cada iteración, hace algún tipo de "paso". La forma del bucle **for** es:

```
for(inicialización; Expresión condicional; paso)
sentencia
```

Cualquiera de las expresiones *inicialización*, *expresión condicional* o *paso* puede estar vacía. Dicha expresión se evalúa antes de cada iteración, y en cuanto el resultado sea falso, la ejecución continuará en la línea siguiente a la sentencia **for**. Al final de cada iteración se ejecuta *paso*.

Los bucles **for** suelen utilizarse para crear contadores:

```
//: c03:ListaCaracteres.java
// Muestra el funcionamiento del bucle "for" listando
// todos los caracteres ASCII.

public class ListaCaracteres {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // Limpiar pantalla ANSI
      System.out.println(
        "valor: " + (int)c +
        " caracter: " + c);
  }
} ///:~
```

Fíjese en que la variable **c** está definida en el punto en que se usa, dentro de la expresión de control del bucle **for**, en vez de al principio del bloque delimitado por la llave de apertura. El ámbito de **c** es la expresión controlada por el **for**.

Los lenguajes procedurales tradicionales como C requieren que todas las variables se definan al principio de un bloque, de forma que cuando el compilador cree un bloque, pueda asignar espacio para esas variables. En Java y C++ es posible diseminar las declaraciones de variables a lo largo del bloque, definiéndolas en el momento en que son necesarias. Esto permite un estilo de codificación más natural y hace que el código sea más fácil de entender.

Se puede definir múltiples variables dentro de una sentencia **for**, pero deben ser del mismo tipo:

```
for(int i = 0, j =1;
  i < 10 && j ¡= 11;
  i++, j++)
  /* cuerpo del bucle for */
```

La definición **int** de la sentencia **for** cubre tanto a **i** como a **j**. La habilidad de definir variables en expresiones de control se limita al bucle **for**. No se puede utilizar este enfoque con cualquiera de las otras sentencias de selección o iteración.

El operador coma

Anteriormente en este capítulo, dije que el *operador* coma (no el *separador* coma, que se usa para separar definiciones y parámetros de funciones) sólo tiene un uso en Java: en la expresión de control de un bucle **for**. Tanto en la inicialización como en las porciones de "paso" de las expresiones de control, se tiene determinado número de sentencias separadas por comas, y estas sentencias se evaluarán secuencialmente. El fragmento de bloque previo utiliza dicha capacidad. He aquí otro ejemplo:

```
//: c03:OperadorComa.java
public class OperadorComa {
     public static void main(String[] args) {
for(int i = 1, j = i + 10; i < 5;
i++, j = i * 2) {
System.out.println("i= " + i + " j= " + j);
     }
   }
} ///:~
```

He aquí la salida:

```
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
```

Se puede ver que tanto en la inicialización, como en las porciones de "paso" se evalúan las sentencias en orden secuencial. Además, la porción de inicialización puede tener cualquier número de definiciones *de un tipo*.

break y continue

Dentro del cuerpo de cualquier sentencia de iteración también se puede controlar el flujo del bucle utilizando **break** y **continue**. **Break** sale del bucle sin ejecutar el resto de las sentencias del bucle. **Continue** detiene la ejecución de la iteración actual y vuelve al principio del bucle para comenzar la siguiente iteración.

Este programa muestra ejemplos de **break** y **continue** dentro de bucles **for** y **while**:

```
//: c03:BreakYContinue.java
// Muestra el funcionamiento de las palabras clave break y continue.
public class BreakYContinue {
     public static void main(String[] args) {
for(int i = 0; i < 100; i++) {
if(i == 74) break; // Sale del bucle for
  if(i % 9 != 0) continue; // Siguiente iteración
  System.out.println(i);
}
  int i = 0;
  // Un "bucle infinito":
  while(true) {
    i++;
    int j = i * 27;
  if(j == 1269) break; // Sale del bucle
  if(i % 10 != 0) continue; // Parte superior del bucle
  System.out.println(i);
     }
  }
} ///:~
```

En el bucle **for** el valor de **i** nunca llega a 100 porque la sentencia **break** rompe el bucle cuando **i** vale 74. Normalmente, el **break** sólo se utilizaría de esta manera si no se supiera cuándo va a darse la condición de terminación. La sentencia **continue** hace que la ejecución vuelva a la parte superior del bucle de iteración (incrementando por consiguiente la **i**) siempre que **i** no sea totalmente divisible por 9. Cuando lo es, se imprime el valor.

La segunda porción muestra un "bucle infinito" que debería, en teoría, continuar para siempre. Sin embargo, dentro del bucle hay una sentencia **break** que romperá el bucle y saldrá de él. Además, se verá que la sentencia **continue** vuelve a la parte de arriba del bucle sin completar el resto.

(Por consiguiente la impresión se da en el segundo bucle sólo cuando el valor de **i** es divisible por 10.) La salida es:

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

El valor 0 se imprime porque 0 % 9 da 0.

Una segunda forma de hacer un bucle infinito es escribir **for(;;)**. El compilador trata tanto a **while (true),** como a **for(;;)** de la misma manera, de forma que cualquiera que se use en cada caso, no es más que una cuestión de gusto.

El infame "goto"

La palabra clave **goto** ha estado presente en los lenguajes de programación desde los comienzos. Sin duda, el **goto** era la génesis del control de los programas en el lenguaje ensamblador: "Si se da la condición A, entonces saltar aquí, sino, saltar ahí." Si se lee el código ensamblador generado al final por cualquier compilador, se verá que el control del programa contiene muchos saltos. Sin embargo, un **goto** es un salto al nivel de código fuente, y eso es lo que le ha traído tan mala reputación. Si un programa siempre salta de un punto a otro, ¿no hay forma de reorganizarlo de manera que el flujo de control no dé tantos saltos? **goto** cayó en desgracia con la publicación del famoso artículo "El Goto considerado dañino"[3], de Edsger Dijkstra, y desde entonces, la prohibición del goto ha sido un deporte popular, con los partidarios de la palabra clave repudiada buscando guarida.

Como es típico en situaciones como ésta, el terreno imparcial es el más fructífero. El problema no es el uso del **goto**, sino el uso excesivo de **goto** —en raras ocasiones el **goto** es de hecho la mejor manera de estructurar el flujo del programa.

Aunque **goto** es una palabra reservada en Java, no se utiliza en el lenguaje; Java no tiene **goto**. Sin embargo, tiene algo que se parece un poco a un salto atado vinculado a las palabras clave **break** y **continue**. No es un salto sino más bien una forma de romper una sentencia de iteración. El motivo por el que aparece muy a menudo en discusiones relacionadas con el **goto,** es que utiliza el mismo mecanismo: una etiqueta.

Una etiqueta es un identificador seguido de dos puntos, como ésta:

```
etiqueta1:
```

El *único* sitio en el que una etiqueta es útil en Java es justo antes de una sentencia de iteración. Y eso significa *justo* antes —no hace ningún bien poner cualquier otra sentencia entre la etiqueta y la iteración. Y la única razón para poner una etiqueta antes de una iteración es si se va a anidar otra iteración o un "switch" dentro. Eso es porque las palabras **break** y **continue** únicamente interrumpirán normalmente al bucle actual, pero cuando se usan con una etiqueta, interrumpirán a los bucles hasta donde exista la etiqueta:

```
etiqueta1:
iteracion-externa {
     iteracion-interna {
  //...
```

[3] Nota del traductor: "Goto considered harmful".

```
      break; //1
      //...
      continue; //2
      //...
      continue etiqueta1; //3
      //...
      break etiqueta1; //4
    }
  }
```

En el caso 1, el **break** rompe la iteración interna, pasando a la iteración exterior. En el caso 2, el **continue** hace volver al principio de la iteración interna. Pero en el caso 3, el **continue etiqueta1** rompe tanto la iteración interna, *como* la externa, retrocediendo hasta **etiqueta1**. Posteriormente, de hecho, continúa la iteración, pero empezando en la iteración exterior. En el caso 4, el **break etiqueta1** también rompe el bucle haciendo volver hasta **etiqueta1**, pero no vuelve a entrar en la iteración. De hecho, rompe ambas iteraciones.

He aquí un ejemplo de utilización de bucles **for**:

```
//: c03:ForEtiquetado.java
// El bucle "for etiquetado" de Java.

public class ForEtiquetado {
public static void main(String[] args) {
      int i = 0;
      externo: // Aquí no puede haber sentencias
      for(; true ;) { // bucle infinito
      interno: // Aquí no puede haber sentencias
      for(; i < 10; i++) {
  visualizar("i = " + i);
  if(i == 2) {
  visualizar("continuar");
    continue;
          }
  if(i == 3) {
  visualizar("salir");
  i++; // En caso contrario i
  // no se incrementa nunca.
  break;
          }
  if(i == 7) {
  visualizar("continuar el externo");
  i++; // En caso contrario i
  // no se incrementa nunca.
  continue externo;
          }
```

```
  if(i == 8) {
  visualizar("salir externo");
  break externo;
          }
  for(int k = 0; k < 5; k++) {
  if(k == 3) {
  prt("continuar el interno");
  continue interno;
            }
          }
        }
     }
  // Aquí no se puede hacer break o continue
  // a etiquetas
  }
  static void visualizar(String s) {
  System.out.println(s);
  }
} ///:~
```

Este ejemplo usa el método **visualizar( )** que ha sido definido en los otros ejemplos.

Nótese que **break** sale del bucle **for**, y que la expresión de incremento no se da hasta acabar de pasar por el bucle **for**. Dado que **break** se salta la expresión e incremento, el incremento se da directamente en el caso de **i==3**. La sentencia **continuar externo** en el caso de **i==7** va también a la parte superior del bucle, y se salta también el incremento, por lo que también se incrementa directamente.

He aquí la salida:

```
i = 0
continuar el interno
i = 1
continuar el interno
i = 2
continuar
i = 3
salir
i = 4
continuar el interno
i = 5
continuar el interno
i = 6
continuar el interno
i = 7
continuar el externo
```

```
i = 8
salir externo
```

Si no fuera por la sentencia **break externo**, no habría manera de salir del bucle externo desde el bucle interno, dado que **break**, por sí misma puede romper únicamente el bucle más interno. (Y lo mismo ocurre con **continue**.)

Por supuesto, en los casos en los que salir de un bucle implique también salir del método, uno puede usar simplemente un **return**.

He aquí una demostración de sentencias etiquetadas **break** y **continue** con bucles **while**:

```
//: c03:WhileEtiquetado.java
// El bucle "while etiquetado" de Java.

public class WhileEtiquetado {
  public static void main(String[] args) {
  int i = 0;
  externo:
  while(true) {
    visualizar("Bucle while externo");
    while(true) {
      i++;
    visualizar("i = " + i);
      if(i == 1) {
    visualizar("continuar");
        continue;
}
  if(i == 3) {
    visualizar("Continuar externo");
    continue externo;
}
  if(i == 5) {
    visualizar("salir");
    break;
}
  if(i == 7) {
    visualizar("break externo");
    break externo;
        }
      }
    }
  }
  static void visualizar(String s) {
  System.out.println(s);
  }
} ///:~
```

Las mismas reglas son ciertas para **while**:

1. Un **continue** sin más va hasta el comienzo del bucle más interno, y continúa.
2. Un **continue** etiquetado va a la etiqueta, y vuelve a entrar en el bucle situado justo después de la etiqueta.
3. Un **break** "abandona" el bucle.
4. Un **break** etiquetado abandona el final del bucle marcado por la etiqueta.

La salida de este método lo deja claro:

```
Bucle while externo
i = 1
continuar
i = 2
i = 3
continuar externo
Bucle while externo
i = 4
i = 5
salir
Bucle while externo
i = 6
i = 7
salir externo
```

Es importante recordar que la *única* razón para usar etiquetas en Java es cuando se tienen bucles anidados, y se quiere utilizar sentencias **break** o **continue** a través de más de un nivel de anidamiento.

En el artículo "El goto considerado dañino" de Dijkstra, se ponen objeciones a las etiquetas, no al goto en sí. Dijkstra observó que el número de errores tiende a incrementarse con el número de etiquetas que haya en un programa. Las etiquetas y las sentencias goto hacen difícil el análisis estático, puesto que introducen ciclos en el grafo de ejecución de los programas. Fíjese que las etiquetas de Java no tienen este problema, pues están limitadas a su ubicación, y no pueden ser utilizadas para transferir el control de forma directa. También es interesante tener en cuenta que éste es el caso en el que una característica de un lenguaje se convierte en más interesante, simplemente restringiendo el poder de la propia sentencia.

switch

La orden **switch** suele clasificarse como *sentencia de selección*. La sentencia **switch** selecciona de entre fragmentos de código basados en el valor de una expresión entera. Es de la forma:

```
switch (selector-entero) {
  case valor-entero1 : sentencia; break;
```

```
  case valor-entero2 : sentencia; break;
  case valor-entero3 : sentencia; break;
  case valor-entero4 : sentencia; break;
  case valor-entero5 : sentencia; break;
       // ...
  default : sentencia;
}
```

El *selector entero* es una expresión que produce un valor entero. El **switch** compara el resultado de *selector entero* con cada *valor entero*. Si encuentra un valor que coincida, ejecuta la sentencia (simple o compuesta) correspondiente. Si no encuentra ninguna coincidencia, ejecuta la *sentencia* **default**.

Observese en la definición anterior que cada **case** acaba con **break**, lo que causa que la ejecución salte al final del cuerpo de la sentencia **switch**. Ésta es la forma convencional de construir una sentencia **switch**, pero el **break** es opcional. Si no se pone, se ejecutará el código de las sentencias "case" siguientes, hasta encontrar un **break**. Aunque este comportamiento no suele ser el deseado, puede ser útil para un programador experimentado. Hay que tener en cuenta que la última sentencia, la que sigue a **default**, no tiene **break** porque la ejecución llega hasta donde le hubiera llevado el **break**. Se podría poner un **break** al final de la sentencia **default** sin que ello causara ningún daño, si alguien lo considerara importante por razones de estilo.

La sentencia **switch** es una forma limpia de implementar una selección de múltiples caminos (por ejemplo, seleccionar un camino de entre cierto número de caminos de ejecución diferentes), pero requiere de un selector que se evalúe a un valor como **int** o **char**. Si se desea utilizar, por ejemplo, una cadena de caracteres o un número de coma flotante como selector, no se podrá utilizar una sentencia **switch**. En el caso de tipos no enteros, es necesario utilizar una serie de sentencias **if**.

He aquí un ejemplo que crea letras al azar y determina si se trata de vocales o consonantes:

```
//: c03:VocalesYConsonantes.java
// Demuestra el funcionamiento de la sentencia switch.

public class VocalesYConsonantes {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      char c = (char)(Math.random() * 26 + 'a');
      System.out.print(c + ": ");
      switch(c) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
                System.out.println("vocal");
                break;
      case 'y':
```

```
        case 'w':
                    System.out.println(
                      "A veces una vocal");
                    break;
        default:
                    System.out.println("consonante");
        }
      }
    }
} ///:~
```

Dado que **Math.random( )** genera un valor entre 0 y 1, sólo es necesario multiplicarlo por el límite superior del rango de números que se desea producir (26 para las letras del alfabeto) y añadir un desplazamiento para establecer el límite inferior.

Aunque aquí parece que se está haciendo un **switch** con un carácter, esta sentencia está usando, de hecho, el valor entero del carácter. Los caracteres entre comillas simples de las sentencias **case** también producen valores enteros que se usan para las comparaciones.

Fíjese cómo las sentencias **case** podrían "apilarse" unas sobre otras para proporcionar varias coincidencias para un fragmento de código particular. También habría que ser conscientes de que es esencial poner la sentencia **break** al final de un caso particular, de otra manera, el control simplemente irá descendiendo, pasando a ejecutar el **case** siguiente.

Detalles de cálculo

La sentencia

char c = (char)(Math.random( ) * 26 +'a');

merece una mirada más detallada. **Math.random( )** produce un **double**, por lo que se convierte el valor 26 a **double** para llevar a cabo la multiplicación, que también produce un **double**. Esto significa que debe convertirse la **'a'** a **double** para llevar a cabo la suma. El resultado **double** se vuelve a convertir en **char** con un **molde**.

¿Qué es lo que hace la conversión a **char**? Es decir, si se tiene el valor 29,7 y se convierte a **char,** ¿cómo se sabe si el valor resultante es 30 o 29? La respuesta a esta pregunta se puede ver en este ejemplo:

```
//: c03:ConvertirNumeros.java
// ¿Qué ocurre cuando se convierte un float
// o un double a un valor entero?

public class ConvertirNumeros {
  public static void main(String[] args) {
    double
      encima = 0.7,
      debajo = 0.4;
```

```
      System.out.println("encima: " + encima);
      System.out.println("debajo: " + debajo);
      System.out.println(
        "(int)encima: " + (int)encima);
      System.out.println(
        "(int)debajo: " + (int)debajo);
      System.out.println(
        "(char)('a' + encima): " +
        (char)('a' + encima));
      System.out.println(
        "(char)('a' + debajo): " +
        (char)('a' + debajo));
  }
} ///:~
```

La salida es:

```
encima: 0.7
debajo: 0.4
(int)encima: 0
(int)debajo: 0
(char)('a' + encima) = a
(char)('a' + debajo) = a
```

Por lo que la respuesta es que si se hace una conversión de un **float** o un **double** a un valor entero lo truncará.

Hay una segunda cuestión que concierne a **Math.random( )**. ¿Produce un valor de cero a uno, incluyendo o excluyendo al valor '1'? En el lingo matemático ¿es (0, 1) o [0, 1], o (0, 1] o [0, 1)? (El corchete significa "incluye" mientras que el paréntesis significa "excluye".) De nuevo, la solución la puede proporcionar un programa de prueba:

```
//: c03:LimitesAleatorios.java
// ¿Produce Math.random()0.0 y 1.0?

public class LimitesAleatorios {
  static void uso() {
    System.out.println("Utilizacion: \n\t" +
      "LimitesAleatorios inferior\n\t" +
      "LimitesAleatorios superior");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length != 1) uso();
    if(args[0].equals("inferior")) {
      while(Math.random() != 0.0)
        ; // Seguir intentándolo
```

```
        System.out.println("Produjo 0.0!");
      }
      else if(args[0].equals("superior")) {
        while(Math.random() != 1.0)
          ; // Seguir intentandolo
        System.out.println("Produjo 1.0!");
      }
      else
        uso();
  }
} ///:~
```

Para ejecutar el programa, se teclea una línea de comandos como:

```
java LimitesAleatorios inferior
```

o

```
java LimitesAleatorios superior
```

En ambos casos nos vemos forzados a romper el programa manualmente, de forma que da la sensación de que **Math.random( )** nunca produce ni 0,0 ni 1,0. Pero éste es el punto en el que un experimento así puede defraudar. Si se considera[4] que hay al menos 2^{62} fracciones **double** distintas entre 0 y 1, la probabilidad de alcanzar cualquier valor experimentalmente podría superar el tiempo de vida de un computador o incluso el de la persona que realiza la prueba. Resulta que 0,0 *está* incluido en la salida de **Math.random( )**. O, en el lingo de las matemáticas es [0, 1).

Resumen

Este capítulo concluye el estudio de los aspectos fundamentales que aparecen en la mayoría de los lenguajes de programación: cálculo, precedencia de operadores, conversión de tipos, y selección e iteración. Ahora estamos listos para empezar a dar pasos y acercarse al mundo de la programación

[4] Chuck Allison escribe: "El número total de números en el sistema de números en coma flotante es **2(M–m+1) b^(p–1) +1**, donde **b** es la base (generalmente 2), **p** es la precisión (dígitos de la mantisa), **M** es el exponente mayor, y **m** es el exponente menor. IEEE 754 utiliza:

M = 1023, m = -1022, p = 53, b = 2

por lo que el número total de números es

2(1023+1022+1)2^52

= 2((2^10-1)+(2^10-1)2^52

= (2^10-1)2^54

=2^64 - 2^54

La mitad de estos números (los correspondientes a los exponentes del rango [-1022, 1] son menores a 1 en magnitud (tanto positivos como negativos), por lo que 1/4 de esa expresión, o 2^62 – 2^52+1 (aproximadamente 2^62) está en el rango [0, 1). Véase mi artículo en

*http://www.freshsources.com/*1995006.htm (final del texto).

orientada a objetos. El siguiente capítulo cubrirá los aspectos importantes de la inicialización y limpieza de objetos, seguido del esencial concepto de ocultación de información en el capítulo siguiente.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com.*

1. Hay dos expresiones en la sección denominada "precedencia" de este capítulo. Poner estas expresiones en un programa y demostrar que producen resultados diferentes.
2. Poner los métodos temario( ) y alternativo( ) en un programa que funcione.
3. Poner los métodos prueba( ) y prueba2( ) de las secciones "if-else" y "return" en un programa que funcione.
4. Escribir un programa que imprima valores de 1 a 100.
5. Modificar el Ejercicio 4, de forma que el programa exista utilizando la palabra clave break en el valor 47. Intentar hacerlo usando return en vez de break.
6. Escribir una función que reciba como parámetros dos cadenas de texto, y use todas las comparaciones lógicas para comparar ambas cadenas e imprimir los resultados. Para el caso de == y !=, llevar a cabo también las pruebas de equals( ). En main( ), llamar a la función con varios objetos String distintos.
7. Escribir un programa que genere 25 valores enteros al azar. Para cada valor, utilizar una sentencia if-then-else para clasificarlo como mayor, menor o igual que un segundo valor generado al azar.
8. Modificar el Ejercicio 7, de forma que el código esté dentro de un bucle while "infinito". Después se ejecutará hasta que se interrumpa desde el teclado (generalmente presionando Control-C).
9. Escribir un programa que use dos bucles for anidados y el operador módulo (%) para detectar e imprimir números primos (números enteros que no son divisibles por otro número que no sean ellos mismos o 1).
10. Crear una sentencia switch que escriba un mensaje en cada caso, e introducirla en un bucle for que pruebe cada caso. Poner un break después de cada caso y probarlo. A continuación, quitar las sentencias break y ver qué ocurre.

4: Inicialización y limpieza

A medida que progresa la revolución computacional, la programación "insegura" se ha convertido en uno de los mayores culpables del encarecimiento de la programación.

Dos de estos aspectos de seguridad son la *inicialización* y la *limpieza*. Muchos de los fallos que se dan en C ocurren cuando el programador olvida inicializar una variable. Esto es especialmente habitual con las bibliotecas, cuando los usuarios no saben cómo inicializar un componente de una biblioteca, o incluso cuándo deben hacerlo. La limpieza o eliminación es un problema especial porque es fácil olvidarse de un elemento una vez que ya no se utiliza, puesto que ya no tiene importancia. Por consiguiente, los recursos que ese elemento utilizaba quedan reservados y es fácil acabar quedándose sin recursos (y el más importante, la memoria).

C++ introdujo el concepto de *constructor*, un método especial invocado automáticamente en la creación de un objeto. Java también adoptó el constructor, y además tiene un recolector de basura que libera automáticamente recursos de memoria cuando dejan de ser utilizados. Este capítulo examina los aspectos de inicialización y eliminación, y su soporte en Java.

Inicialización garantizada con el constructor

Es posible imaginar la creación de un método denominado **inicializar( )** para cada clase que se escriba. El nombre se debe invocar antes de utilizar el objeto. Por desgracia, esto significa que el usuario debe recordar llamar al método. En Java, el diseñador de cada clase puede garantizar que se inicialice cada objeto proporcionando un método especial llamado *constructor*. Si una clase tiene un constructor, Java llama automáticamente al constructor cuando se crea un objeto, antes de que los usuarios puedan siquiera pensar en poner sus manos en él. Por consiguiente, la inicialización queda garantizada.

El siguiente reto es cómo llamar a este método. Hay dos aspectos. El primero es que cualquier nombre que se use podría colisionar con un nombre que nos gustaría utilizar como miembro en una clase. El segundo es que dado que el compilador es el responsable de invocar al constructor, debe saber siempre qué método invocar. La solución de C++ parece la mejor y más lógica, por lo que se utiliza también en Java: el nombre del constructor es el mismo que el nombre de la clase. Tiene sentido que un método así se invoque automáticamente en la inicialización.

He aquí hay una clase con un constructor simple:

```
//: c04:ConstructorSimple.java
// Muestra de un constructor simple.
```

```
class Roca {
  Roca() { // Éste es el constructor
    System.out.println("Creando Roca");
  }
}

public class ConstructorSimple {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Roca();
  }
} ///:~
```

Ahora, al crear un objeto:

```
new Roca();
```

se asigna almacenamiento y se invoca al constructor. Queda garantizado que el objeto será inicializado de manera adecuada antes de poder poner las manos sobre él.

Fíjese que el estilo de codificación de hacer que la primera letra de todos los métodos sea minúscula no se aplica a los constructores, dado que el nombre del constructor debe coincidir *exactamente* con el nombre de la clase.

Como cualquier método, el constructor puede tener parámetros para permitir especificar *cómo* se crea un objeto. El ejemplo de arriba puede cambiarse sencillamente de forma que el constructor reciba un argumento:

```
//: c04:ConstructorSimple2.java
// Los constructores pueden tener parámetros.

class Roca2 {
  Roca2(int i) {
    System.out.println(
      "Creando la roca numero " + i);
  }
}

public class ConstructorSimple2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Roca2(i);
  }
} ///:~
```

Los parámetros del constructor proporcionan un medio para pasar parámetros a la inicialización de un objeto. Por ejemplo, si la clase **Arbol** tiene un constructor que toma un número entero que indica la altura del árbol, crearíamos un objeto **Arbol** como éste:

```
Arbol a = new Arbol(12); // Un árbol de 12 metros
```

Si **Arbol(int)** es el único constructor, entonces el compilador no permitirá crear un objeto **Arbol** de ninguna otra forma.

Los constructores eliminan un montón de problemas y simplifican la lectura del código. En el fragmento de código anterior, por ejemplo, no se verá ninguna llamada explícita a ningún método **inicializar( )** que esté conceptualmente separado, por definición. En Java, la definición e inicialización son conceptos que están unidos —no se puede tener uno sin el otro.

El constructor es un tipo inusual de método porque no tiene valor de retorno. Esto es muy diferente al valor de retorno **void**, en el que el método no devuelve nada pero se sigue teniendo la opción de hacer que devuelva algo más. Los constructores no devuelven nada y no es necesario tener ninguna opción. Si hubiera un valor de retorno, y si se pudiera seleccionar el propio, el compilador, de alguna manera, necesitaría saber qué hacer con ese valor de retorno.

Sobrecarga de métodos

Uno de los aspectos más importantes de cualquier lenguaje de programación es el uso de los nombres. Al crear un objeto, se da un nombre a cierta región de almacenamiento. Un método es un nombre que se asigna a una acción. Al utilizar nombres para describir el sistema, se crea un programa más fácil de entender y de modificar por la gente. Es como escribir en prosa —la meta es comunicarse con los lectores.

Para hacer referencia a objetos y métodos se usan nombres. Los nombres bien elegidos hacen más sencillo que todos entiendan un código.

Surge un problema cuando se trata de establecer una correspondencia entre el concepto de matiz del lenguaje humano y un lenguaje de programación. A menudo, la misma palabra expresa varios significados —se ha *sobrecargado*. Esto es útil, especialmente cuando incluye diferencias triviales. Se dice "lava la camisa", "lava el coche" y "lava el perro". Sería estúpido tener que decir "lavaCamisas la camisa", "lavaCoche el coche" y "lavaPerro el perro" simplemente para que el que lo escuche no tenga necesidad de intentar distinguir entre las acciones que se llevan a cabo. La mayoría de los lenguajes humanos son redundantes, por lo que incluso aunque se te olviden unas pocas palabras, se sigue pudiendo entender. No son necesarios identificadores únicos —se puede deducir el significado del contexto.

La mayoría de los lenguajes de programación (C en particular) exigen que se tenga un identificador único para cada función. Así, no se podría tener una función llamada **print( )** para imprimir enteros si existe ya otra función llamada **print( )** para imprimir decimales —cada función requiere un nombre único.

En Java (y C++) otros factores fuerzan la sobrecarga de los nombres de método: el constructor. Dado que el nombre del constructor está predeterminado por el nombre de la clase, sólo puede haber un nombre de constructor. Pero ¿qué ocurre si se desea crear un objeto de más de una manera? Por ejemplo, suponga que se construye una clase que puede inicializarse a sí misma de manera estándar o leyendo información de un archivo. Se necesitan dos constructores, uno que no tome argumentos (el constructor *por defecto*, llamado también constructor *sin parámetros*), y otro que tome como parámetro un **String**, que es el nombre del archivo con el cual inicializar el objeto. Ambos son constructores, por lo que deben tener el mismo nombre —el nombre de la clase. Por consiguiente, la *sobrecarga de métodos* es esencial para permitir que se use el mismo nombre de métodos con distintos tipos de parámetros. Y aunque la sobrecarga de métodos es una necesidad para los constructores, es bastante conveniente y se puede usar con cualquier método.

He aquí un ejemplo que muestra métodos sobrecargados, tanto constructores como ordinarios:

```
//: c04:Sobrecarga.java
// Muestra de sobrecarga de métodos
// tanto constructores como ordinarios.
import java.util.*;

class Arbol {
  int altura;
  Arbol() {
    visualizar("Plantando un retoño");
    altura = 0;
  }
  Arbol(int i) {
    visualizar("Creando un nuevo arbol que tiene "
        + i + " metros de alto");
    altura = i;
  }
  void info() {
    visualizar("El arbol tiene " + altura
        + " metros de alto");
  }
  void info(String s) {
    visualizar(s + ": El arbol tiene "
        + altura + " metros de alto");
  }
  static void visualizar(String s) {
    System.out.println(s);
  }
}

public class sobrecarga {
  public static void main(String[] args) {
```

```
      for(int i = 0; i < 5; i++) {
        Arbol t = new Arbol(i);
        t.info();
        t.info("metodo sobrecargado");
      }
      // Constructor sobrecargado:
      new Arbol();
    }
} ///:~
```

Se puede crear un objeto **Arbol,** bien como un retoño, sin argumentos, o como una planta que crece en un criadero, con una altura ya existente. Para dar soporte a esto, hay dos constructores, uno que no toma argumentos (a los constructores sin argumentos se les llama *constructores por defecto*[1]), y uno que toma la altura existente.

Podríamos también querer invocar al método **info( )** de más de una manera. Por ejemplo, con un parámetro **String** si se tiene un mensaje extra para imprimir, y sin él si no se tiene nada más que decir. Parecería extraño dar dos nombres separados a lo que es obviamente el mismo concepto. Afortunadamente, la sobrecarga de métodos permite usar el mismo nombre para ambos.

Distinguir métodos sobrecargados

Si los métodos tienen el mismo nombre, ¿cómo puede saber Java qué método se debe usar en cada caso? Hay una regla simple: cada método sobrecargado debe tomar una única lista de tipos de parámetros.

Si se piensa en esto por un segundo, tiene sentido: ¿de qué otra forma podría un programador distinguir entre dos métodos que tienen el mismo nombre si no fuera por los tipos de parámetros?

Incluso las diferencias en el orden de los parámetros son suficientes para distinguir ambos métodos: (Aunque normalmente este enfoque no es necesario, pues produce un código difícil de mantener.)

```
//: c04:OrdenSobrecarga.java
// Sobrecarga basada en el
// orden de los parámetros.

public class OrdenSobrecarga {
  static void print(String s, int i) {
    System.out.println(
      "cadena: " + s +
      ", entero: " + i);
  }
```

[1] En algunos documentos sobre Java de Sun, por el contrario, se refieren a éstos con el poco elegante pero descriptivo nombre de "constructores sin parámetros". El término "constructor por defecto" se ha usado durante muchos años, por lo que será el que utilizaremos.

```
   static void print(int i, String s) {
      System.out.println(
         "Entero: " + i +
         ", Cadera: " + s);
   }
   public static void main(String[] args) {
      print("Primero cadena", 11);
      print(99, "Primero entero");
   }
} ///:~
```

Ambos métodos **print( )** tienen los mismos argumentos, pero distinto orden, y eso es lo que los hace diferentes.

Sobrecarga con tipos primitivos

Un tipo primitivo puede ser promocionado automáticamente de un tipo menor a otro mayor, y esto puede ser ligeramente confuso si se combina con la sobrecarga. El ejemplo siguiente demuestra lo que ocurre cuando se pasa un tipo primitivo a un método sobrecargado:

```
//: c04:SobrecargaPrimitivo.java
// Promoción de tipos primitivos y sobrecarga.

public class SobrecargaPrimitivo {
   // los booleanos no pueden convertirse automáticamente
   static void visualizar(String s) {
      System.out.println(s);
   }

   void f1(char x) { visualizar("f1(char)"); }
   void f1(byte x) { visualizar("f1(byte)"); }
   void f1(short x) { visualizar("f1(short)"); }
   void f1(int x) { visualizar("f1(int)"); }
   void f1(long x) { visualizar("f1(long)"); }
   void f1(float x) { visualizar("f1(float)"); }
   void f1(double x) { visualizar("f1(double)"); }

   void f2(byte x) { visualizar("f2(byte)"); }
   void f2(short x) { visualizar("f2(short)"); }
   void f2(int x) { visualizar("f2(int)"); }
   void f2(long x) { visualizar("f2(long)"); }
   void f2(float x) { visualizar("f2(float)"); }
   void f2(double x) { visualizar("f2(double)"); }

   void f3(short x) { visualizar("f3(short)"); }
```

```
void f3(int x) { visualizar("f3(int)"); }
void f3(long x) { visualizar("f3(long)"); }
void f3(float x) { visualizar("f3(float)"); }
void f3(double x) { visualizar("f3(double)"); }

void f4(int x) { visualizar("f4(int)"); }
void f4(long x) { visualizar("f4(long)"); }
void f4(float x) { visualizar("f4(float)"); }
void f4(double x) { visualizar("f4(double)"); }

void f5(long x) { visualizar("f5(long)"); }
void f5(float x) { visualizar("f5(float)"); }
void f5(double x) { visualizar("f5(double)"); }

void f6(float x) { visualizar("f6(float)"); }
void f6(double x) { visualizar("f6(double)"); }

void f7(double x) { visualizar("f7(double)"); }

void pruebaValoresConstante() {
  visualizar("Probando con el 5");
  f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
}
void pruebaChar() {
  char x = 'x';
  visualizar("parametro char:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void pruebaByte() {
  byte x = 0;
  visualizar("parametro byte:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void pruebaShort() {
  short x = 0;
  visualizar("parametro short:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void pruebaInt() {
  int x = 0;
  visualizar("parametro int:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void pruebaLong() {
  long x = 0;
```

```
      visualizar("parametro long:");
      f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
   }
   void pruebaFloat() {
      float x = 0;
      visualizar("parametro float:");
      f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
   }
   void pruebaDouble() {
      double x = 0;
      visualizar("parametro double:");
      f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
   }
   public static void main(String[] args) {
      SobrecargaPrimitivo p =
         new SobrecargaPrimitivo();
      p.pruebaValoresConstante();
      p.pruebaChar();
      p.pruebaByte();
      p.pruebaShort();
      p.pruebaInt();
      p.pruebaLong();
      p.pruebaFloat();
      p.pruebaDouble();
   }
} ///:~
```

Si se observa la salida de este programa, se verá que el valor constante 5 se trata como un **int**, de forma que si hay disponible un método sobrecargado que tome un **int**, será el utilizado. En todos los demás casos, si se tiene un tipo de datos menor al parámetro del método, ese tipo de dato será promocionado. Un dato de tipo **char** produce un efecto ligeramente diferente, pues si no encuentra una coincidencia exacta de **char**, se promociona a **int**.

¿Qué ocurre si el parámetro es *mayor* que el que espera el método sobrecargado? La respuesta la proporciona una modificación del programa anterior:

```
//: c04:Degradacion.java
// Degradación de tipos primitivos y sobrecarga.

public class Degradacion {
   static void visualizar(String s) {
      System.out.println(s);
   }

   void f1(char x) { visualizar("f1(char)"); }
   void f1(byte x) { visualizar("f1(byte)"); }
```

```
  void f1(short x) { visualizar("f1(short)"); }
  void f1(int x) { visualizar("f1(int)"); }
  void f1(long x) { visualizar("f1(long)"); }
  void f1(float x) { visualizar("f1(float)"); }
  void f1(double x) { visualizar("f1(double)"); }

  void f2(char x) { visualizar("f2(char)"); }
  void f2(byte x) { visualizar("f2(byte)"); }
  void f2(short x) { visualizar("f2(short)"); }
  void f2(int x) { visualizar("f2(int)"); }
  void f2(long x) { visualizar("f2(long)"); }
  void f2(float x) { visualizar("f2(float)"); }

  void f3(char x) { visualizar("f3(char)"); }
  void f3(byte x) { visualizar("f3(byte)"); }
  void f3(short x) { visualizar("f3(short)"); }
  void f3(int x) { visualizar("f3(int)"); }
  void f3(long x) { visualizar("f3(long)"); }

  void f4(char x) { visualizar("f4(char)"); }
  void f4(byte x) { visualizar("f4(byte)"); }
  void f4(short x) { visualizar("f4(short)"); }
  void f4(int x) { visualizar("f4(int)"); }

  void f5(char x) { visualizar("f5(char)"); }
  void f5(byte x) { visualizar("f5(byte)"); }
  void f5(short x) { visualizar("f5(short)"); }

  void f6(char x) { visualizar("f6(char)"); }
  void f6(byte x) { visualizar("f6(byte)"); }

  void f7(char x) { visualizar("f7(char)"); }

  void pruebaDouble() {
    double x = 0;
    visualizar("parametro double:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
  }
  public static void main(String[] args) {
    Degradacion p = new Degradacion();
    p.pruebaDouble();
  }
} ///:~
```

Aquí, los métodos toman valores primitivos de menor tamaño. Si el parámetro es de mayor tamaño es necesario *convertir* el parámetro al tipo necesario poniendo entre paréntesis el nombre del tipo. Si no se hace esto, el compilador mostrará un mensaje de error.

Uno debería ser consciente de que ésta es una *conversión reductora*, que significa que podría conllevar una pérdida de información durante la conversión. Éste es el motivo por el que el compilador obliga a hacerlo —para marcar la conversión reductora.

Sobrecarga en los valores de retorno

Es común preguntarse "¿Por qué sólo los nombres de las clases y las listas de parámetros de los métodos? ¿Por qué no distinguir entre métodos basados en sus valores de retorno?" Por ejemplo, estos dos métodos, que tienen el mismo nombre y parámetros, se distinguen fácilmente el uno del otro:

```
void f() {}
int f() {}
```

Esto funciona bien cuando el compilador puede determinar de manera inequívoca el significado a partir del contexto, como en **int x = f( )**. Sin embargo, se puede llamar a un método e ignorar el valor de retorno; a esto se le suele llamar *invocar a un método por su efecto lateral*, dado que no hay que tener cuidado sobre el valor de retorno y sí desear los otros efectos de la llamada al método. Por tanto, si se llama al método de la siguiente manera:

```
f();
```

¿cómo puede determinar Java qué **f( )** invocar? ¿Y cómo podría alguien más que lea el código verlo también? Debido a este tipo de problemas, no se pueden usar los tipos de valores de retorno para distinguir los métodos sobrecargados.

Constructores por defecto

Como se mencionó anteriormente, un constructor por defecto (un constructor sin parámetros) es aquél que no tiene parámetros, y se utiliza para crear un "objeto básico". Si se crea una clase que no tiene constructores, el compilador siempre creará un constructor por defecto. Por ejemplo:

```
//: c04:ConstructorPorDefecto.java

class Pajaro {
  int i;
}

public class ConstructorPorDefecto {
  public static void main(String[] args) {
    Pajaro nc = new Pajaro(); // ¡por defecto!
  }
} ///:~
```

La línea

```
new Pajaro();
```

crea un objeto nuevo e invoca a la función constructor, incluso aunque ésta no se haya definido explícitamente. Sin ella no habría ningún método a invocar para construir el objeto. Sin embargo, si se define algún constructor (con o sin parámetros) el compilador *no* creará uno automáticamente:

```
class Arbusto {
      Arbusto (int i) {}
      Arbusto (double d) {}
}
```

Ahora, si se escribe

```
new Arbusto();
```

el compilador se quejará por no poder encontrar un constructor que coincida. Es como si no se pusiera ningún constructor, y el compilador dice "Debes necesitar *algún* constructor, por lo que crearé uno". Pero si escribes un constructor, el compilador dice "Has escrito un constructor por lo que ya sabes lo que estás haciendo; si no hiciste un constructor por defecto es porque no lo necesitabas".

La palabra clave **this**

Si se tiene dos objetos del mismo tipo llamados **a** y **b**, nos pondríamos preguntar cómo es que se puede invocar a un método **f( )** para ambos objetos:

```
class Platano { void f (int i) { /* ... */ } }
Platano a = new Platano(), b = new Platano();
a.f(1);
b.f(2);
```

Si sólo hay un método llamado **f( )**, ¿cómo puede este método saber si está siendo invocado para el objeto **a** o **b**?

Para permitir la escritura de código con una sintaxis adecuada orientada a objetos en la que "se envía un menaje a un objeto", el compilador se encarga del código clandestino. Hay un primer parámetro secreto que se pasa al método **f( )**, y ese parámetro es la referencia al objeto que está siendo manipulado. Por tanto, las dos llamadas a método anteriores, se convierten en algo parecido a:

```
Platano.f(a,1);
Platano.f(b,2);
```

Esto es interno y uno no puede escribir estas expresiones y hacer que el compilador las acepte, pero da una idea de lo que está ocurriendo.

Supóngase que uno está dentro de un método y que desea conseguir la referencia al objeto actual. Dado que esa referencia se pasa *de forma secreta* al compilador, no hay un identificador para él. Sin embargo, para este propósito hay una palabra clave: **this**. Esta palabra clave —que puede usarse sólo dentro de un método— produce la referencia al objeto por el que se ha invocado al método. Uno puede tratar esta referencia como cualquier otra referencia a un objeto. Hay que recordar que si se está invocando a un método de una clase desde dentro de un método de esa misma clase, no es necesario utilizar **this**; uno puede simplemente invocar al método. Por consiguiente, se puede decir:

```
Class Albaricoque{
     void tomar() { /* ... */ }
     void deshuesar() { tomar(); /* ... */}
}
```

Dentro de **deshuesar( )**, uno *podría* decir **this.tomar( ),** pero no hay ninguna necesidad. El compilador lo hace automáticamente. La palabra clave **this** sólo se usa para aquellos casos especiales en los que es necesario utilizar explícitamente la referencia al objeto actual. Por ejemplo, se usa a menudo en sentencias **return** cuando se desea devolver la referencia al objeto actual:

```
//: c04:Hoja.java
// Utilización simple de la palabra clave "this".

public class Hoja {
  int i = 0;
  Hoja incrementar() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Hoja x = new Hoja();
    x.incrementar().incrementar().incrementar().print();
  }
} ///:~
```

Dado que **incrementar( )** devuelve la referencia al objeto actual, a través de la palabra clave **this**, pueden ejecutarse múltiples operaciones con el mismo objeto.

Invocando a constructores desde constructores

Cuando se escriben varios constructores para una clase, hay veces en las que uno quisiera invocar a un constructor desde otro para evitar la duplicación de código. Esto se puede lograr utilizando la palabra clave **this**.

Normalmente, cuando se dice **this**, tiene el sentido de "este objeto" o "el objeto actual", y por sí mismo produce la referencia al objeto actual. En un constructor, la palabra clave **this** toma un significado diferente cuando se le da una lista de parámetros: hace una llamada explícita al constructor que coincida con la lista de parámetros. Por consiguiente, hay una manera directa de llamar a otros constructores:

```
//: c04:Flor.java
// Invocación a constructores con "this".

public class Flor {
  int numeroPetalos = 0;
  String s = new String("null");
  Flor(int petalos) {
    numeroPetalos = petalos;
    System.out.println(
      "Constructor w/ parametro entero solo, Numero de petalos = "
      + numeroPetalos);
  }
  Flor(String ss) {
    System.out.println(
      "Constructor w/ parametro cadera solo, s=" + ss);
    s = ss;
  }
  Flor(String s, int petalos) {
    this(petalos);
//!    this(s); // ¡No se puede invocar dos!
    this.s = s; // Otro uso de "this"
    System.out.println("cadena y entero Parámetros");
  }
  Flor() {
    this("Hola", 47);
    System.out.println(
      "constructor por defecto (sin parametros)");
  }
  void print() {
//!    this(11); // ;No dentro de un no-constructor!
    System.out.println(
      "Numero de Petalos = " + numeroPetalos + " s = "+ s);
  }
  public static void main(String[] args) {
    Flor x = new Flor();
    x.print();
  }
} ///:~
```

El constructor **Flor(String s, int petalos)** muestra quese puede invocar a un constructor utilizando **this**, pero no a dos. Además, la llamada al constructor debe ser la primera cosa que se haga o se obtendrá un mensaje de error del compilador.

El ejemplo también muestra otra manera de ver el uso de **this**. Dado que el nombre del parámetro **s** y el nombre del atributo **s** son el mismo, hay cierta ambigüedad. Se puede resolver diciendo **this.s** para referirse al dato miembro. A menudo se verá esta forma en código Java, que también se usa en muchas partes de este libro.

En **print( )** se puede ver que el compilador no permite invocar a un constructor desde dentro de otro método que no sea un constructor.

El significado de estático (static)

Teniendo en cuenta la palabra clave this, uno puede comprender completamente qué significa hacer un método **estático**. Significa que no hay un **this** para ese método en particular. No se puede invocar a métodos no **estático** desde dentro de métodos **estáticos**[2] (aunque al revés sí que es posible), y se puede invocar al método **estático** de la propia clase, sin objetos. De hecho, esto es principalmente el fin de un método **estático**. Es como si se estuviera creando el equivalente a una función global (de C). La diferencia es que las funciones globales están prohibidas en Java, y poner un método **estático** dentro de una clase permite que ésta acceda a otros métodos **estáticos** y a campos **estáticos**.

Hay quien discute que los métodos **estáticos** no son orientados a objetos, puesto que tienen la semántica de una función global; con un método **estático** no se envía un mensaje a un objeto, puesto que no hay **this**. Esto probablemente es un argumento justo, y si uno acaba usando *un montón* de métodos estáticos, seguro que tendrá que replantearse su estrategia. Sin embargo, los métodos **estáticos** son pragmáticos y hay veces en las que son genuinamente necesarios, por lo que el hecho de que sean o no "POO pura" se deja para los teóricos. Sin duda, incluso Smalltalk tiene un equivalente en sus "métodos de clase ".

Limpieza: finalización y recolección de basura

Los programadores conocen la importancia de la inicialización, pero a menudo se les olvida la importancia de la limpieza. Después de todo, ¿quién necesita eliminar un **int**? Pero con las bibliotecas, dejar que un objeto simplemente "se vaya" una vez que se ha acabado con él, no es siempre seguro. Por supuesto, Java tiene el recolector de basura para recuperar la memoria de los objetos que ya no se usan. Considere ahora un caso muy inusual. Supóngase que los objetos asignan memoria "especial" sin utilizar **new**. El recolector de basura sólo sabe liberar la memoria asignada con **new**, por lo que ahora no sabrá cómo liberar esa memoria "especial" del objeto. Para hacer frente a este caso, Java proporciona un método denominado **finalize( )** que se puede definir en cada clase. He aquí cómo se *supone* que funciona. Cuando el recolector de basura está preparado para liberar el espacio de almacenamiento uti-

[2] El único caso en el que esto podría ocurrir es si se pasa una referencia a un objeto dentro del método **estático**. Después, a través de la referencia (que ahora es **this**) se puede invocar a métodos no **estáticos** y acceder a campos no **estáticos**. Pero generalmente si se desea hacer algo así, simplemente se hará un método ordinario no **estático**.

lizado por el objeto, primero invocará a **finalize( )**, y sólo recuperará la memoria del objeto durante la pasada del recolector de basura. Por tanto, si se elige usar **finalize( )**, éste te proporciona la habilidad de llevar a cabo alguna limpieza importante *a la vez que la recolección de basura.*

Éste es un error potencial de programación porque algunos programadores, especialmente los de C++, podrían confundir **finalize( )** con el *destructor* de C++, que es una función que siempre se invoca cuando se destruye un objeto. Pero es importante distinguir entre C++ y Java en este caso, pues en C++ *los objetos siempre se destruyen* (en un programa sin errores), mientras que los objetos de Java no siempre son eliminados por el recolector. O, dicho de otra forma:

La recolección de basura no es destrucción

Si se recuerda esto, se evitarán los problemas. Lo que significa es que si hay alguna actividad que debe llevarse a cabo antes de que un objeto deje de ser necesario, hay que llevar a cabo esa actividad por uno mismo. Java no tiene un destructor o un concepto similar, por lo que hay que crear un método ordinario para hacer esta limpieza. Por ejemplo, supóngase que en el proceso de creación de un objeto, éste se dibuja a sí mismo en la pantalla. Si no se borra explícitamente esta imagen de la pantalla, podría ser que éste no se elimine nunca. Si se pone algún tipo de funcionalidad eliminadora dentro de **finalize( )**, si un objeto es eliminado por el recolector de basura, la imagen será eliminada en primer lugar de la pantalla, pero si no lo es, la imagen permanecerá. Por tanto, un segundo punto a recordar es:

Los objetos podrían no ser eliminados por el recolector de basura

Uno podría averiguar que el espacio de almacenamiento de un objeto nunca se libera porque el programa nunca llega a quedarse sin espacio de almacenamiento. Si el programa se completa y el recolector de basura nunca llega a ejecutarse para liberar el espacio de almacenamiento de ningún objeto, éste será devuelto por completo al sistema operativo en el momento en que acaba el programa. Esto es bueno, porque el recolector de basura tiene algo de sobrecarga, y si nunca se ejecuta, no hay que incurrir en ese gasto.

¿Para qué sirve **finalize( )**?

Uno podría pensar en este punto que no deberíamos utilizar **finalize( )** como un método de limpieza de propósito general. ¿Cómo de bueno es?

Un tercer punto para recordar es:

La recolección de basura sólo tiene que ver con la memoria

Es decir, la única razón para la existencia de un recolector de basura, es recuperar la memoria que un programa ha dejado de utilizar. Por tanto, cualquier actividad asociada a la recolección de basura, especialmente el método **finalize( )** debe estar relacionada también sólo con la memoria y su desasignación.

¿Significa esto que si un objeto contiene otros objetos **finalize( )** debería liberar explícitamente esos objetos? Pues no —el recolector de basura cuida de la liberación de toda la memoria de los objetos independientemente de cómo se creará el objeto. Resulta que la necesidad de **finalize( )** se limita a casos especiales, en los que un objeto puede reservar espacio de almacenamiento de forma distinta a la creación de un objeto. Pero, podríamos pensar: en Java todo es un objeto, así que ¿cómo puede ser?

Parecería que **finalize( )** tiene sentido debido a la posibilidad de que se haga algo de estilo C, asignando memoria utilizando un mecanismo distinto al normal de Java. Esto puede ocurrir principalmente a través de *métodos nativos*, que son la forma de invocar a código no-Java desde Java. (Los métodos nativos se discuten en el Apéndice B.) C y C++ son los únicos lenguajes actualmente soportados por los métodos nativos, pues dado que pueden llamar a subprogramas escritos en otros lenguajes, pueden efectivamente invocar a cualquier cosa. Dentro del código no-Java, se podría invocar a la familia de funciones de **malloc( )** de C para asignar espacio de almacenamiento, provocando una pérdida de memoria. Por supuesto, **free( )** es una función de C y C++, por lo que sería necesario invocarla en un método nativo desde el **finalize( )**.

Después de leer esto, probablemente se tendrá la idea de que no se usará mucho **finalize( )**. Es correcto: no es el sitio habitual para que ocurra una limpieza normal. Por tanto, ¿dónde debería llevarse a cabo la limpieza normal?

Hay que llevar a cabo la limpieza

Para eliminar un objeto, el usuario debe llamar a un método de limpieza en el punto en el que se desee. Esto suena bastante directo, pero colisiona un poco con el concepto de destructor de C++. En este lenguaje, se destruyen todos los objetos. O mejor dicho, *deberían* eliminarse todos los objetos. Si se crea el objeto C++ como local (por ejemplo, en la pila —lo cual no es posible en Java), la destrucción se da al cerrar la llave del ámbito en el que se ha creado el objeto. Si el objeto se creó usando **new** (como en Java) se llama al destructor cuando el programador llame al operador **delete** de C++ (que no existe en Java). Si el programador de C++ olvida invocar a **delete**, no se llama nunca al destructor, y se tiene un fallo de memoria, y además las otras partes del objeto no se borran nunca. Este tipo de fallo suele ser muy difícil de localizar.

Por el contrario, Java no permite crear objetos locales —siempre hay que usar **new**. Pero en Java, no hay un "eliminar" al que invocar para liberar el objeto, dado que el recolector de basura se encarga de liberar el espacio de almacenamiento. Por tanto, desde un punto de vista simplista, se podría decir que por culpa del recolector de basura, Java no tiene destructor. Se verá a medida que se vaya avanzando en el libro, que la presencia de un recolector de basura no elimina la necesidad de, o la utilidad de los destructores (y no se debería invocar a **finalize( )** directamente, pues ésta no es la solución más adecuada). Si se desea llevar a cabo algún tipo de limpieza distinta a la liberación de espacio de almacenamiento, hay que *seguir* llamando explícitamente al método apropiado en Java, que es el equivalente al destructor de C++ , sea o no lo más conveniente.

Una de las cosas para las que puede ser útil **finalize( )** es para observar el proceso de recolección de basura. El ejemplo siguiente resume las descripciones anteriores del recolector de basura:

```
//: c04:Basura.java
// Demostración de recolector de
// basura y finalización

class Silla {
  static boolean ejecrecol = false;
  static boolean f = false;
  static int creadas = 0;
  static int finalizadas = 0;
  int i;
  Silla() {
    i = ++creadas;
    if(creadas == 47)
      System.out.println("Creadas 47");
  }
  public void finalize() {
    if(!ejecrecol) {
      // La primera vez se invoca a finalize():
      ejecrecol = true;
      System.out.println(
        "Comenzando a finalizar tras haber creado " +
        creadas + " sillas");
    }
    if(i == 47) {
      System.out.println(
        "Finalizando la silla #47, " +
        "Poniendo el indicada que evita la creacion de mas sillas");
      f = true;
    }
    finalizadas++;
    if(finalizadas >= creadas)
      System.out.println(
        "Las " + finalizadas + " han sido finalizadas");
  }
}

public class Basura {
  public static void main(String[] args) {
    // Mientras no se haya puesto el flag,
    // hacer sillas y cadenas de texto:
    while(!Silla.f) {
      new Silla();
      new String("Coger cspacio");
    }
    System.out.println(
      "Despues de haber creado todas las sillas:\n" +
```

```
        "creadas en total = " + Silla.creadas +
        ", finalizadas total = " + Silla.finalizadas);
      // Parámetros opcionales fueran la recolección
      // de basura y finalización:
      if(args.length > 0) {
        if(args[0].equals("rec") ||
           args[0].equals("todo")) {
          System.out.println("gc():");
          System.gc();
        }
        if(args[0].equals("finalizar") ||
           args[0].equals("todo")) {
          System.out.println("runFinalization():");
          System.runFinalization();
        }
      }
      System.out.println("adios!");
    }
} ///:~
```

El programa anterior crea muchos objetos **Silla**, y en cierto momento después de que el recolector de basura comience a ejecutarse, el programa deja de crear objetos de tipo **Silla**. Dado que el recolector de basura puede ejecutarse en cualquier momento, uno no sabe exactamente cuando empezará, y hay un indicador denominado **ejecrecol** para indicar si el recolector de basura ha comenzado ya su ejecución o no. Un segundo indicador **f** es la forma de que **Silla** le comunique al bucle **main( )** que debería dejar de hacer objetos. Ambos indicadores se ponen dentro de **finalize( )**, que se invoca durante la recolección de basura.

Otras dos variables **estáticas**, **creadas** y **finalizadas**, mantienen el seguimiento del número de objetos de tipo **Silla** creadas frente al número de finalizadas por el recolector de basura. Finalmente, cada **Silla** tiene su propio (no **estático**) **int i**, por lo que se hace un seguimiento de qué número es. Cuando finalice la **Silla** número 47, el indicador se pone a **true** para detener el proceso de creación de objetos de tipo **Silla**.

Todo esto ocurre en el método **main( )**, en el bucle

```
while(!Silla.f) {
      new Silla( );
      new String("Coger espacio");
}
```

Uno podría preguntarse cómo conseguir finalizar este bucle, dado que no hay nada dentro del bucle que cambie el valor de **Silla.f**. Sin embargo, el proceso **finalize( )** se supone que lo hará cuando finalice el número 47.

La creación de un objeto **String** en cada iteración es simplemente la asignación de almacenamiento extra para animar al recolector de basura a actuar, lo que hará cuando se empiece a poner nervioso por la cantidad de memoria disponible.

Cuando se ejecute el programa, se proporciona un parámetro de línea de comandos que pueden ser "rec", "finalizar" o "todo". El argumento "rec" invocará al método **System.gc( )** (para forzar la ejecución del recolector de basura). La utilización del parámetro "finalizar" invoca a **System.runFinalization( )** que —en teoría— hará que finalicen los objetos que no lo hayan hecho. Y "todo" hace que se llame a los dos métodos.

El comportamiento de este programa y de la versión de la primera edición de este libro muestra que todo lo relacionado con el recolector de basura y la finalización ha evolucionado, habiendo ocurrido mucha de esta evolución detrás del telón. De hecho, para cuando se lea esto, puede que el comportamiento del programa haya vuelto a cambiar.

Si se invoca a **System.gc( )**, se finalizan todos los objetos. Esto no era necesario en el caso de las implementaciones previas del JDK, aunque la documentación decía otra cosa. Además, se verá que no parece haber ninguna diferencia si se invoca o no a **System.runFinalization( )**.

Sin embargo, se verá que sólo si se invoca a **System.gc( )** después de crear y descartar todos los objetos se invocará a todos los finalizadores. Si no se invoca a **System.gc( )**, entonces sólo se finalizan algunos de los objetos. En Java 1.1, se introdujo un método **System.runFinalizersOnExit( )** que hacía que los programas ejecutaran todos los finalizadores al salir, pero el diseño resultó tener errores y se desechó el método. Esto puede ser otro de los motivos por los que los diseñadores de Java siguen dándole vueltas al problema de la recolección de basura y la finalización. Esperamos que este asunto se termine de resolverse en Java 2.

El programa precedente muestra que la promesa de que todos los finalizadores se ejecuten siempre es verdadera, pero sólo uno fuerza explícitamente el que suceda. Si no se fuerza la invocación a **System.gc( )**, se logra una salida como:

```
Creadas 47
Comenzando a finalizar tras haber creado 3486
Finalizando la silla #47
Poniendo el indicador que evita la creacion de mas sillas
Despues de haber creado todas las sillas:
total creadas = 3881, total finalizadas = 2684
adios!
```

Por consiguiente, no se invoca a todos los finalizadores cuando acaba el programa. Si se llama a **System.gc( )**, acabará y destruirá todos los objetos que no estén en uso en ese momento.

Recuérdese que ni el recolector de basura ni la finalización están garantizadas. Si la Máquina Virtual Java (JVM) no está a punto de quedarse sin memoria, entonces (sabiamente) no malgastará tiempo en recuperar memoria mediante el recolector de basura.

La condición de muerto

En general, no se puede confiar en que se invoque a **finalize( )**, y es necesario crear funciones de "limpieza" aparte e invocarlas explícitamente. Por tanto, parece que **finalize( )** solamente es útil para limpiezas oscuras de memoria que la mayoría de programadores nunca usarán. Sin embargo,

hay un uso muy interesante de **finalize( )** que no confía en ser invocada siempre. Se trata de la verificación de la *condición de muerte*[3] de un objeto.

En el momento en que uno deja de estar interesado en un objeto —cuando está listo para ser eliminado— el objeto debería estar en cierto estado en el que su memoria pueda ser liberada de manera segura. Por ejemplo, si el objeto representa un fichero abierto, ese fichero debería ser cerrado por el programador antes de que el objeto sea eliminado por el recolector de basura. Si no se eliminan correctamente ciertas porciones del objeto, se tendrá un fallo en el programa que podría ser difícil de encontrar. El valor de **finalize( )** es que puede usarse para descubrir esta condición, incluso si no se invoca siempre. Si una de las finalizaciones acaba revelando el fallo, se descubre el problema, que es de lo que verdaderamente hay que cuidar.

He aquí un ejemplo simple de cómo debería usarse:

```
//: c04:CondicionMuerte.java
// Utilización de finalize() para detectar un
// objeto que no ha sido eliminado correctamente.

class Libro {
  boolean comprobado = false;
  Libro(boolean comprobar) {
    comprobado = comprobar;
  }
  void correcto() {
    comprobado = false;
  }
  public void finalize() {
    if(comprobado)
      System.out.println("Error: comprobado");
  }
}

public class CondicionMuerte {
  public static void main(String[] args) {
    Libro novela = new Libro(true);
    // Eliminación correcta:
    novela.correcto();
    // Cargarse la referencia, olvidando la limpieza:
    new Libro(true);
    // Forzar la recolección de basura y finalización:
    System.gc();
  }
} ///:~
```

La condición de muerte consiste en que todos los objetos **Libro** supuestamente serán comprobados antes de ser recogidos por el recolector de basura, pero en el método **main( )** un error del pro-

[3] Un término acuñado por Hill Venners *(www.artima.com)* durante un seminario que él y yo impartimos conjuntamente.

gramador no comprueba alguno de los libros. Sin **finalize( )** para verificar la condición de muerte, este error podría ser difícil de encontrar.

Nótese que se usa **System.gc( )** para forzar la finalización (y se debería hacer esto durante el desarrollo del programa para forzar la depuración). Pero incluso aunque no se use, es muy probable descubrir objetos de tipo **Libro** errantes a lo largo de ejecuciones repetidas del programa (asumiendo que el programa asigna suficiente espacio de almacenamiento para hacer que se ejecute el recolector de basura).

Cómo funciona un recolector de basura

Si se realiza en un lenguaje de programación en el que la asignación de objetos en el montículo es cara, hay que asumir naturalmente que el esquema de Java de asignar todo (excepto los datos primitivos) en el montículo es caro.

Sin embargo, resulta que el recolector de basura puede tener un impacto significativo en un *incremento* de la velocidad de creación de los objetos. Esto podría sonar un poco extraño al principio —que la liberación de espacio de almacenamiento afecte a la asignación de espacio— pero es la manera en que trabajan algunas JVM, y significa que la asignación de espacio para objetos del montículo en Java pueda ser casi tan rápida como crear espacio de almacenamiento en la pila en otros lenguajes.

Por ejemplo, se puede pensar que el montículo de C++ es como un terreno en el que cada objeto toma un fragmento de suelo. Puede ser que este espacio sea abandonado tiempo después, y haya que reutilizarlo. En algunas JVM, el montículo de Java es bastante distinto; es más como una cinta transportadora que avanza cada vez que se asigna un nuevo objeto. Esto significa que la asignación de espacio de almacenamiento de los objetos es notoriamente rápida. El "puntero del montículo" simplemente se mueve hacia delante en territorio virgen, así que es exactamente lo mismo que la asignación de pila de C++. (Por supuesto, hay una pequeña sobrecarga por el mantenimiento de espacios, pero no hay nada como buscar espacio de almacenamiento.)

Ahora uno puede observar que el montículo no es, de hecho, una cinta transportadora, pues si se trata como tal podría comenzar eventualmente una paginación excesiva de memoria (que constituye un factor de rendimiento importante), e incluso más tarde la memoria podría agotarse. El truco es que el recolector de basura va paso a paso, y mientras recolecta la basura, compacta todos los objetos de la pila de forma que el resultado es que se ha movido el "puntero del montículo" más cerca del principio de la cinta transportadora y más lejos de un fallo de página. El recolector de basura reorganiza los elementos y hace posible usar un modelo de montículo de alta velocidad y árbol infinito, durante la asignación de espacio de almacenamiento.

Para entender cómo funciona esto es necesario tener una idea un poco mejor de la manera en que funcionan los diferentes esquemas de recolección de basura (GC, *Garbage Collector*). Una técnica simple pero lenta de GC es contar referencias. Esto significa que cada ejemplo tiene un contador de referencias, y cada vez que se adjunte una referencia a un objeto se incrementa en uno el contador de referencias. Cada vez que una referencia cae fuera del ámbito o se pone **null** se decrementa el contador de referencias. Por consiguiente, la gestión de contadores de referencias supone una carga constante y pequeña que se va produciendo durante toda la vida del programa. El recolector de

basura va recorriendo toda la lista de objetos y al encontrar alguno con el contador de referencias a cero, libera el espacio de almacenamiento que tenía asignado. El inconveniente radica en que si los objetos tienen referencias circulares entre sí es posible no encontrar contadores de referencias a cero, que, sin embargo, pueden ser basura. La localización de estos grupos auto-referenciados requiere de una carga de trabajo significativa por parte del recolector de basura. La cuenta de referencias se usa frecuentemente para explicar un tipo de recolección de basura, pero parece no estar implementada en ninguna Máquina Virtual de Java.

En esquemas más rápidos, la recolección de basura no se basa en la cuenta de referencias. Se basa, en cambio, en la idea de que cualquier objeto no muerto podrá recorrerse, realizar una traza en última instancia, hasta una referencia que resida bien en la pila o bien en espacio de almacenamiento estático. La cadena podría atravesar varias capas de objetos. Por consiguiente, si se comienza en la pila y en el área de almacenamiento estático y se van recorriendo todas las referencias, será posible localizar todos los objetos vivos. Por cada referencia que se encuentre, es necesario hacer un recorrido traceo hasta localizar el objeto al que apunta y después seguir todas las referencias a *ese* objeto, recorriendo todos los objetos a los que apunta, etc., hasta haber recorrido toda la red que se originó con la referencia de la pila o del almacenamiento estático. Cada objeto que se recorra debe seguir necesariamente vivo. Fíjese que no hay ningún problema con los grupos auto-referenciados —simplemente no son localizados en el recorrido, trazado, por lo que se considerarán basura automáticamente.

En la aproximación descrita, la Máquina Virtual de Java usa un esquema de recolección de basura adaptativo, y lo que hace con los objetos vivos que encuentra depende de la variante que se haya implementado. Una de estas variaciones es la de *parar-y-copiar*. Esto significa que —por razones que pronto parecerán evidentes— el programa se detiene en primer lugar (este esquema no implica recolección en segundo plano). Posteriormente, cada objeto vivo que se encuentre se copia de un montículo a otro, dejando detrás toda la basura. Además, a medida que se copian los ejemplos al nuevo montículo, se empaquetan de extremo a extremo, compactando por consiguiente el nuevo montículo (y permitiendo recorrer rápida y simplemente el nuevo almacenamiento hasta el final, como se describió previamente).

Por supuesto, cuando se mueve un objeto de un lugar a otro, hay que cambiar todas las referencias que apuntan a ese objeto. Las referencias que vayan del montículo o del área de almacenamiento estática a un objeto pueden cambiarse directamente, pero puede haber otras referencias que apunten a este objeto y que se encuentren más tarde durante la "búsqueda". Éstas se van recomponiendo a medida que se encuentren (podría imaginarse una tabla que establezca una relación entre las direcciones viejas y las nuevas).

También hay dos aspectos que hacen ineficientes a estos denominados "recolectores de copias". El primero es la idea de que son necesarios dos montículos y se maneja por toda la memoria adelante y atrás entre estos dos montículos separados, manteniendo el doble de memoria de la que de hecho se necesita. Algunas Máquinas Virtuales de Java siguen este esquema asignando el montículo por bloques a medida que son necesarios y haciendo simplemente copias de bloques.

El segundo aspecto es la copia. Una vez que el programa se vuelve estable, debería generar poca o ninguna basura. Además de esto, un recolector de copias seguiría copiando toda la memoria de un

sitio a otro, lo que es una pérdida de tiempo y recursos. Para evitar esto, algunas Máquinas Virtuales de Java detectan que no se esté generando nueva basura y pasan a un esquema distinto (ésta es la parte "adaptativa"). Este otro esquema denominado *marcar y barrer*[4], es el que usaban las primeras versiones de la Máquina Virtual de Java de Sun. Para uso general, el esquema de marcar y barrer es bastante lento, pero si se genera poca o ninguna basura es rápido.

El marcar y barrer sigue la misma lógica de empezar rastreando a través de todas las referencias, a partir de la pila y el almacenamiento estático, para encontrar objetos vivos. Sin embargo, cada vez que encuentra un objeto vivo, lo marca poniendo a uno cierto indicador, en vez de recolectarlo. Sólo cuando acaba el proceso de marcado, se da el barrido. Durante el barrido, se liberan los objetos muertos. Sin embargo, no se da ninguna copia, de forma que si el recolector elige recolectar un montículo fragmentado, lo hace reordenando todos los objetos.

El "parar-y-copiar" se refiere a la idea de que este tipo de recolector de basura *no* se hace en segundo plano; sino que, por el contrario, se detiene el programa mientras se ejecuta el recolector de basura. En la documentación de Sun se encuentran muchas referencias a la recolección de basura como un proceso de segundo plano de baja prioridad, pero resulta que el recolector de basura no está implementado así, al menos en las primeras versiones de la Máquina Virtual de Java de Sun. En vez de esto, el recolector de basura de Sun se ejecutaba cuando quedaba poca memoria. Además el marcado y barrido requiere la detención del programa.

Como se mencionó previamente, en la Máquina Virtual de Java aquí descrita, la memoria se asigna por bloques grandes. Si se asigna un objeto grande, éste se hace con un bloque propio. El parar-y-copiar estricto exige copiar todos los objetos vivos del montículo fuente a un montículo nuevo antes de poder liberar el viejo, lo que se traduce en montones de memoria. Con los objetos, el recolector de basura puede en ocasiones usar los bloques muertos para copiar los objetos al ir recolectando. Cada bloque tiene un *contador de generación* para mantener información sobre si está o no vivo. En circunstancias normales, sólo se compactan los bloques creados desde la última recolección. Así se maneja la gran cantidad de objetos temporales de vida corta. Periódicamente, se hace un barrido completo —se siguen sin copiar los objetos grandes, y se copian y compactan todos los bloques que tienen objetos pequeños. La Máquina Virtual de Java monitoriza la eficiencia de la recolección de basura y si se convierte en una pérdida de tiempo porque todos los objetos tienen vida larga, pasa al esquema de marcar-y-barrer. De manera análoga, la Máquina Virtual de Java mantiene un registro del éxito del marcar-y-borrar, y si el montículo comienza a estar fragmentado vuelve de nuevo al parar-y-copiar. Éste es el momento en que interviene la parte "adaptativa", de forma que finalmente se tiene un nombre kilométrico: "Marcado-y-borrado con parada-y-copia adaptativo generacional".

Hay varias técnicas que permiten acelerar la velocidad de la Máquina Virtual de Java. Una especialmente importante se refiere a la operación del cargador y del compilador "justo-a-tiempo" (JIT). Cuando hay que cargar una clase (generalmente, la primera vez que se desea crear un objeto de esa clase), se localiza el fichero **.class** y se lleva a memoria el "codigo byte" de esa clase. En ese momento, un enfoque sería compilar JIT todo el código, pero esto tiene dos inconvenientes: lleva un poco más de tiempo, lo cual, extrapolado a toda la vida del programa puede ser significativo; y aumenta el tamaño del ejecutable (los "códigos byte" son bastante más compactos que el código JIT

---

[3] N. Del traductor: En inglés, *mark and sweep.*

expandido), lo que podría causar paginación, que definitivamente ralentizaría el programa. Un enfoque alternativo lo constituye la *evaluación perezosa*, que quiere decir que el código no se compila JIT hasta que es necesario. Por tanto, el código que no se ejecute nunca será compilado por JIT .

Inicialización de miembros

Java sigue este camino para garantizar que se inicialicen correctamente todas las variables antes de ser utilizadas. En el caso de variables definidas localmente en un método, esta garantía se presenta en forma de error de tiempo de compilación, de forma que si se dice:

```
void f() {
      int i;
      i++;
}
```

se obtendrá un mensaje de error que dice que **i** podría no haber sido inicializada. Por supuesto, el compilador podría haber asignado a **i** un valor por defecto, pero es más probable que se trate de un error del programador, que un valor por defecto habría camuflado. Al forzar al programador a dar un valor de inicialiación es más fácil detectar el fallo.

Sin embargo, las cosas son algo distintas en el caso de atributos de tipo primitivo de una clase. Dado que cualquier método puede inicializar o usar ese dato, podría no ser práctico obligar al usuario a inicializarlo a su valor apropiado antes de usar el dato. Sin embargo, es poco seguro dejarlo con un valor basura, por lo que se garantiza que tendrá un valor inicial. Estos valores pueden verse aquí:

```
//: c04:ValoresIniciales.java
// Muestra los valores iniciales por defecto.

class Medida {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  void escribir() {
    System.out.println(
      "Tipo dato       Valor inicial\n" +
      "boolean          " + t + "\n" +
      "char              [" + c + "] "+ (int)c +"\n"+
      "byte              " + b + "\n" +
      "short             " + s + "\n" +
      "int               " + i + "\n" +
```

```
        "long               " + l + "\n" +
        "float              " + f + "\n" +
        "double             " + d);
  }
}

public class ValoresIniciales {
  public static void main(String[] args) {
    Medida d = new Medida();
    d.escribir();
    /* En este caso también podría decirse:
    new Medida().escribir();
    */
  }
} ///:~
```

La salida del programa será:

```
Tipo dato  Valor inicial
boolean    false
char       [ ] 0
byte       0
short      0
int        0
long       0
float      0.0
double     0.0
```

El valor **char** es un cero, que se imprime como un espacio.

Veremos más adelante que al definir una referencia a un objeto dentro de una clase sin inicializarla a un nuevo objeto, la referencia recibe el valor especial **null** (que es una palabra clave de Java).

Puede incluso verse que, aunque no se especifiquen los valores, se inicializan automáticamente. De esta forma, al menos, no hay amenaza de que se llegue a trabajar con valores sin inicializar.

Especificación de la inicialización

¿Qué ocurre si se quiere dar un valor inicial a una variable? Una manera directa de hacerlo consiste simplemente en asignar el valor al definir la variable en la clase. (Téngase en cuenta que esto no se puede hacer en C++, aunque los novatos en C++ siempre intentan hacerlo). Aquí se han cambiado las definiciones de la clase **Medida** para que proporcionen valores iniciales:

```
class Medida {
  boolean b = true;
  char c = 'x';
```

```
byte b = 47;
short s = 0xff;
int i = 999;
long l = 1;
float f = 3.14f;
double d = 3.14159;
// . . .
```

También se pueden inicializar de la misma manera objetos no primitivos. Si **Profundidad** es una clase, se puede insertar una variable e inicializarla así:

```
class Medida {

  Profundidad o = new Profundidad();
  boolean b = true;
  // . . .
```

Si no se ha dado a **o** un valor inicial e intenta usarlo de cualquier forma, se obtendrá un error de tiempo de ejecución denominado *excepción* (del que se hablará en el Capítulo 10).

Se puede incluso invocar a un método para proporcionar un valor de inicialización:

```
class CInit {
      int i = f();
      //...
}
```

El método puede, por supuesto, tener parámetros, pero éstos no pueden ser sino miembros de la clase que no han sido aún inicializados. Por consiguiente, se puede hacer esto:

```
class CInit {
      int i = f();
      int j = g(i);
      //...
}
```

Pero no se puede hacer esto:

```
class CInit {
      int j = g(i);
      int i = f();
      //...
}
```

Éste es un punto en el que el compilador se *queja*, con razón, del referenciado hacia delante, pues es un error relacionado con el orden de la inicialización y no con la manera de compilar el programa.

Este enfoque de inicialización es simple y directo. Tiene la limitación de que *todo objeto* de tipo **Medida** tendrá los mismos valores de inicialización. Algunas veces esto es justo lo que se necesita, pero otras veces se necesita mayor flexibilidad.

Inicialización de constructores

El constructor puede usarse para llevar a cabo la inicialización, lo que da una flexibilidad mayor en la programación, puesto que se puede invocar a métodos para llevar a cabo acciones en tiempo de ejecución que determinen los tiempos de ejecución. Sin embargo, hay que recordar siempre que no se está excluyendo la inicialización automática, que se da antes de entrar en el constructor. Así, por ejemplo, si se dice:

```
class Contador {
      int i;
      Contador() { i = 7; }
      // . . .
```

se inicializa primero la **i** a 0, y después a 7. Esto es cierto con todos los tipos primitivos y con las referencias a objetos, incluyendo aquéllos a los que se da inicialización explícita en el momento de su definición. Por esta razón, el compilador no intenta forzar la inicialización de elementos del constructor en ningún lugar en concreto, o antes de que se usen —la inicialización ya está garantizada[4].

Orden de inicialización

Dentro de una clase, el orden de inicialización lo determina el orden en que se definen las variables dentro de la clase. Las definiciones de variables pueden estar dispersas a través y dentro de las definiciones de métodos, pero las variables se inicializan antes de invocar a ningún método —incluido el constructor. Por ejemplo:

```
//: c04:OrdenDeInicializacion.java
// Demuestra el orden de inicialización.

// Cuando se invoque al constructor para crear un
// objeto Etiqueta, se verá un mensaje:
class Etiqueta {
  Etiqueta(int marcador) {
    System.out.println("Etiqueta(" + marcador + ")");
  }
}

class Tarjeta {
  Etiqueta t1 = new Etiqueta(1); // Antes del constructor
```

[4] En contraste, C++ tiene la *lista de inicializadores del constructor* que hace que se dé la inicialización antes de entrar en el cuerpo del constructor, y se fuerza para los objetos. Ver *Thinking in C++*, 2.ª edición (disponible en el CD ROM de este libro, y en *http://www.BruceEckel.com*).

```
  Tarjeta() {
    // Indicar que estamos en el constructor:
    System.out.println("Tarjeta()");
    t3 = new Etiqueta(33); // Reiniciar t3
  }
  Etiqueta t2 = new Etiqueta(2); // Después del constructor
  void f() {
    System.out.println("f()");
  }
  Etiqueta t3 = new Etiqueta(3); // Al final
}

public class OrdenDeInicializacion {
  public static void main(String[] args) {
    Tarjeta t = new Tarjeta();
    t.f(); // Muestra que se ha acabado la construcción
  }
} ///:~
```

En **Tarjeta**, la definición de los objetos **Etiqueta** se han dispersado intencionadamente para probar que todos se inicializarán antes de que se llegue a entrar al constructor u ocurra cualquier otra cosa. Además, **t3** se reinicia dentro del constructor. La salida es:

```
Etiqueta(1)
Etiqueta(2)
Etiqueta(3)
Tarjeta()
Etiqueta(33)
f()
```

Por consiguiente, la referencia **t3** se inicializa dos veces, una antes y otra durante la llamada al constructor. (El primer objeto se desecha, de forma que posteriormente podrá ser eliminado por el recolector de basura.) Esto podría parecer ineficiente a primera vista, pero garantiza una inicialización correcta —¿Qué ocurriría si se definiera un constructor sobrecargado que *no* inicializara **t3** y no hubiera una inicialización "por defecto" para **t3** en su definición?

Inicialización de datos estáticos

Cuando los datos son **estáticos** ocurre lo mismo; si se trata de un dato primitivo y no se inicializa, toma los valores iniciales estándares de los tipos primitivos. Si se trata de una referencia a un objeto, es **null,** a menos que se cree un objeto nuevo al que se asocie la referencia.

Si se desea realizar una inicialización en el momento de la definición, ocurre lo mismo que con los no **estáticos**. Sólo hay un espacio de almacenamiento para un dato **estático** independientemente de cuántos objetos se creen. Pero las dudas surgen cuando se inicializa el espacio de almacenamiento de un dato **estático**. Un ejemplo puede aclarar esta cuestión:

```
//: c04:InicializacionStatic.java
// Especificando los valores iniciales en una
// definición de clase.

class Bolo {
  Bolo(int marcador) {
    System.out.println("Bolo(" + marcador + ")");
  }
  void f(int marcador) {
    System.out.println("f(" + marcador + ")");
  }
}

class Mesa {
  static Bolo b1 = new Bolo(1);
  Mesa() {
    System.out.println("Mesa()");
    b2.f(1);
  }
  void f2(int marcador) {
    System.out.println("f2(" + marcador + ")");
  }
  static Bolo b2 = new Bolo(2);
}

class Armario {
  Bolo b3 = new Bolo(3);
  static Bolo b4 = new Bolo(4);
  Armario() {
    System.out.println("Armario()");
    b4.f(2);
  }
  void f3(int marcador) {
    System.out.println("f3(" + marcador + ")");
  }
  static Bolo b5 = new Bolo(5);
}

public class InicializacionStatic {
  public static void main(String[] args) {
    System.out.println(
      "Creando nuevo Armario() en el método main");
    new Armario();
    System.out.println(
      "Creando nuevo Armario() en el método main");
```

```
        new Armario();
        t2.f2(1);
        t3.f3(1);
    }
    static Mesa t2 = new Mesa();
    static Armario t3 = new Armario();
} ///:~
```

Bolo permite ver la creación de una clase, y **Mesa** y **Armario** crean miembros **estáticos** de **Bolo** dispersos por sus definiciones de clases. Fíjese que **Armario** crea un **Bolo** no **estático** antes de las definiciones **estáticas**. La salida muestra lo que ocurre:

```
Bolo(1)
Bolo(2)
Mesa()
f(1)
Bolo(4)
Bolo(5)
Bolo(3)
Armario()
f(2)
Creando nuevo Armario() el método main
Bolo(3)
Armario()
f(2)
Creando nuevo Armario() el método main
Bolo(3)
Armario()
f(2)
f2(1)
f3(1)
```

La inicialización **estática** sólo se da si es necesaria. Si no se crea un objeto **Mesa** y nunca se hace referencia a **Mesa.b1** o **Mesa.b2**, los objetos estáticos de tipo **Bolo b1** y **b2** no se crearán nunca. Sin embargo, se inicializan sólo cuando se cree el *primer* objeto **Mesa** (o se dé el primer acceso **estático**). Después de eso, los objetos **estáticos** no se reinician.

Se inicializan primero los objetos **estáticos**, si todavía no han sido inicializados durante la creación anterior de un objeto, y posteriormente los objetos no estáticos. Se puede ver la prueba de esto en la salida del programa anterior.

Es útil resumir el proceso de creación de un objeto. Considérese una clase llamada **Perro**:

1. La primera vez que se cree un objeto de tipo **Perro**, *o* la primera vez que se acceda a un método **estático** o un campo **estático** de la clase **Perro**, el intérprete de Java debe localizar **Perro.class**, que lo hace buscando a través de las trayectorias de clases.

2. Al cargar **Perro.class** (creando un objeto **Class**, del que se hablará más adelante), se ejecutan todos sus inicializadores **estáticos**. Por consiguiente, la inicialización sólo tiene lugar una vez, al cargar el objeto **Class** la primera vez.

3. Cuando se crea un **new Perro( )**, el proceso de construcción de un objeto **Perro** asigna, en primer lugar, el espacio de almacenamiento suficiente para un objeto **Perro** del montículo.

4. Este espacio de almacenamiento se pone a cero, poniendo automáticamente todos los datos primitivos del objeto **Perro** con sus valores por defecto (cero para los números y su equivalente para los **boolean** o **char**) y las referencias a **null**.

5. Se ejecuta cualquier inicialización que se dé en el momento de la definición de campos.

6. Se ejecutan los constructores. Como se verá en el Capítulo 6, esto podría implicar de hecho una cantidad de actividad considerable, especialmente cuando esté involucrada la herencia .

Inicialización estática explícita

Java permite agrupar todas las inicializaciones **estáticas** dentro de una "cláusula de construcción **estática**" (llamada a veces *bloque estático*) dentro de una clase. Tiene la siguiente apariencia:

```
class Cuchara {
      static int i;
      static {
            i = 47;
      }
      // . . .
```

Parece un método, pero es simplemente la palabra clave **static** seguida de un cuerpo de método. Este código, como otras inicializaciones **estáticas**, se ejecuta sólo una vez, la primera vez que se cree un objeto de esa clase *o* la primera vez que se acceda a un miembro **estático** de esa clase (incluso si nunca se llega a hacer un objeto de esa clase). Por ejemplo:

```
//: c04:StaticExplicito.java
// Inicialización explícita estática
// con la cláusula "static".

class Taza {
  Taza(int marcador) {
    System.out.println("Taza(" + marcador + ")");
  }
  void f(int marcador) {
    System.out.println("f(" + marcador + ")");
  }
}

class Tazas {
```

```
    static Taza c1;
    static Taza c2;
    static {
        c1 = new Taza(1);
        c2 = new Taza(2);
    }
    Tazas() {
        System.out.println("Tazas()");
    }
}

public class StaticExplicito {
    public static void main(String[] args) {
        System.out.println("Dentro de main()");
        Tazas.c1.f(99);  // (1)
    }
    // static Tazas x = new Tazas();  // (2)
    // static Tazas y = new Tazas();  // (2)
} ///:~
```

Los inicializadores **estáticos** de **Tazas** se ejecutan cuando se da el acceso al objeto **estático c1** en la línea marcada (1), si la línea (1) se marca como un comentario, y se quita el signo de comentario de las líneas marcadas como (2). Si tanto (1) como (2) se consideran comentarios, la inicialización **estática** de **Tazas** no se realizará nunca. Además, no importa si una o las dos líneas marcadas (2) dejan de ser comentarios; la inicialización sólo ocurre una vez.

Inicialización de instancias no estáticas

Java proporciona una sintaxis similar para la inicialización de variables no **estáticas** de cada objeto. He aquí un ejemplo:

```
//: c04:Jarras.java
// Java "Inicialización de Instancias."

class Jarra {
    Jarra(int marcador) {
        System.out.println("Jarra(" + marcador + ")");
    }
    void f(int marcador) {
        System.out.println("f(" + marcador + ")");
    }
}

public class Jarras {
    Jarra c1;
    Jarra c2;
```

```
    {
      c1 = new Jarra(1);
      c2 = new Jarra(2);
      System.out.println("c1 y c2 inicializadas");
    }
    Jarras() {
      System.out.println("Jarras()");
    }
    public static void main(String[] args) {
      System.out.println("Dentro de main()");
      Jarras x = new Jarras();
    }
} ///:~
```

Se puede ver que la cláusula de inicialización de instancias:

```
{
      c1 = new Jarra(1);
      c2 = new Jarra(2);
      System.out.println("c1 y c2 inicializadas");
}
```

tiene exactamente la misma apariencia que la cláusula de inicialización estática excepto porque no está la palabra clave **static**. Esta sintaxis es necesaria para dar soporte a la inicialización de clases *internas anónimas* (ver Capítulo 8).

Inicialización de arrays

La inicialización de arrays en C suele ser fuente de errores y tediosa. C++ usa la *inicialización agregada* para hacerla más segura[5]. Java no tiene "agregados" como C++, puesto que en Java todo es un objeto. Tiene arrays, y éstos se soportan con la inicialización de arrays.

Un array es simplemente una secuencia, bien de objetos o bien de datos primitivos, todos del mismo tipo, empaquetados juntos bajo un único identificador. Los arrays se definen y utilizan con el *operador de indexación* entre corchetes **[ ]**. Para definir un array simplemente hay que colocar corchetes vacíos seguidos del nombre del tipo de datos:

```
int [] a1;
```

También se puede poner los corchetes tras el identificador para lograr exactamente el mismo significado:

```
int a1[];
```

[5] *Thinking in C++*, 2.ª edición, para obtener una descripción completa de la inicialización agregada.

Esto satisface las expectativas de los programadores de C y C++. El estilo anterior, sin embargo, es probablemente una sintaxis más sensata, puesto que dice que el tipo es "un array de **int**". Éste será el estilo que se use en este libro.

El compilador no permite especificar el tamaño del array. Esto nos devuelve al aspecto de las "referencias". Todo lo que se tiene en este momento es una referencia a un array, para el que no se ha asignado espacio de almacenamiento. Para crear espacio de almacenamiento para el array es necesario escribir una expresión de inicialización. En el caso de los arrays, la inicialización puede aparecer en cualquier lugar del código, pero puede usarse un tipo especial de expresión de inicialización que debe situarse en el mismo lugar en que el que se crea el array. Esta inicialización especial es un conjunto de valores encerrados entre llaves. Es el compilador el que se encarga de la asignación de espacio (el equivalente a usar **new**). Por ejemplo:

```
int [ ] a1 = { 1, 2, 3, 4, 5 };
```

Por tanto ¿por qué puede definirse una referencia a un array sin un array?

```
int [ ] a2;
```

Bien, es posible asignar un array a otro en Java, por lo que puede decirse:

```
a2 = a1;
```

Lo que se está haciendo realmente es copiar una referencia, como se demuestra a continuación:

```
//: c04:Arrays.java
// Arrays de datos primitivos.

public class Arrays {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i]++;
    for(int i = 0; i < a1.length; i++)
      System.out.println(
        "a1[" + i + "] = " + a1[i]);
  }
} ///:~
```

Puede verse que se da a **a1** un valor de inicialización mientras que a **a2**, no; **a2** se asigna más tarde —en este caso, a otro array.

Aquí hay algo nuevo: todos los arrays tienen un miembro intrínseco (bien sean arrays de objetos o arrays de tipos primitivos) por el que se puede preguntar —pero no modificar— para saber cuántos elementos hay en el array. Este miembro es **length**. Dado que los arrays en Java, como en C y C++, empiezan a contar desde elemento 0, el elemento más lejano que se puede indexar es **length - 1**.

Si se sale de rango, no produce error, siendo esto la fuente de muchos errores graves. Sin embargo, Java le protege de esos problemas originando un error en tiempo de ejecución (una *excepción*, el tema del Capítulo 10) al intentar acceder más allá de los límites. Por supuesto, la comprobación de todos los accesos a arrays supone tiempo y código, y no hay manera de desactivarse, lo que significa que los accesos a arrays podrían ser una fuente de ineficiencia en un programa si se dan en una situación crítica. Los diseñadores de Java pensaron que este sacrificio merecía la pena en aras de la seguridad de Internet y la productividad del programador.

¿Qué ocurre si al escribir el programa se desconocen cuántos elementos son necesarios que tenga el array? Simplemente se utiliza **new** para crear elementos del array. Aquí, **new** funciona incluso aunque se esté creando un array de datos primitivos (**new** no creará datos primitivos que no sean elementos de un array):

```
//: c04:NuevoArray.java
// Creando arrays con new.
import java.util.*;

public class NuevoArray {
  static Random aleatorio = new Random();
  static int pAleatorio(int modulo) {
    return Math.abs(aleatorio.nextInt()) % modulo + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pAleatorio(20)];
    System.out.println(
      "longitud de = " + a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println(
        "a[" + i + "] = " + a[i]);
  }
} ///:~
```

Dado que el tamaño del array se elige al azar (utilizando el método **pAleatorio( )**) está claro que la creación del array se está dando en tiempo de ejecución. Además, se verá en la salida de este programa que los elementos del array de tipos primitivos se inicializan automáticamente a valores "vacíos". (En el caso de valores numéricos y **carácter**, este valor es cero, y en el caso de los **boolean**, es **false**.)

Por supuesto, el array también podría haberse definido e inicializado en la misma sentencia:

```
int [] a = new int[pAleatorio(20)];
```

Si se está tratando con un array de objetos no primitivos, siempre es necesario usar **new**. Aquí, vuelve a surgir el tema de las referencias porque lo que se crea es un array de referencias. Considérese el tipo **Integer**, que es una clase y no un tipo primitivo:

```
//: c04:ObjetoClaseArray.java
// Creando un array de objetos no primitivos.
```

```
import java.util.*;

public class ObjetoClaseArray {
   static Random aleatorio = new Random();
   static int pAleatorio(int modulo) {
      return Math.abs(aleatorio.nextInt()) % modulo + 1;
   }
   public static void main(String[] args) {
      Integer[] a = new Integer[pAleatorio(20)];
      System.out.println(
         "longitud de a = " + a.length);
      for(int i = 0; i < a.length; i++) {
         a[i] = new Integer(pAleatorio(500));
         System.out.println(
            "a[" + i + "] = " + a[i]);
      }
   }
} ///:~
```

Aquí, incluso tras llamar a **new** para crear el array:

```
Integer[] a = new Integer[pAleatorio(20)];
```

se trata sólo de un array de referencias, y no se completa la inicialización hasta que se inicializa la propia referencia creando un nuevo objeto **Integer**:

```
a[i] = new Integer(pAleatorio(500));
```

Si se olvida crear el objeto, sin embargo, se obtiene una excepción en tiempo de ejecución al intentar leer la localización vacía del array.

Eche un vistazo a la creación del objeto **String** dentro de las sentencias de impresión. Puede observarse que la referencia al objeto **Integer** se convierte automáticamente para producir un **String** que representa el valor dentro del objeto.

También es posible inicializar el array de objetos utilizando la lista encerrada entre llaves. Hay dos formas:

```
//: c04:InicializacionArray.java
// Inicialización de arrays.

public class InicializacionArray {
   public static void main(String[] args) {
      Integer[] a = {
         new Integer(1),
         new Integer(2),
         new Integer(3),
      };
```

```
    Integer[] b = new Integer[] {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
} ///:~
```

Esto es útil en ocasiones, pero es más limitado, pues se determina el tamaño del array en tiempo de compilación. La coma final de la lista de inicializadores es opcional. (Esta característica permite un mantenimiento más sencillo de listas largas.)

La segunda forma de inicializar arrays proporciona una sintaxis adecuada para crear y llamar a métodos que pueden producir el mismo efecto que las *listas de parámetros variables* de C (conocidas en este lenguaje como "parametros-variables"). Éstas pueden incluir una cantidad de parámetros desconocida además de tipos desconocidos. Dado que todas las clases se heredan en última instancia de la clase raíz común **Object** (un tema del que se aprenderá más a medida que progrese el libro), se puede crear un método que tome un array de **Object** e invocarlo así:

```
//: c04:ParametrosVariables.java
// Utilizando la sintaxis de arrays para crear
// listas de parámetros variables.

class A { int i; }

public class ParametrosVariables {
  static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    f(new Object[] {
        new Integer(47), new ParametrosVariables(),
        new Float(3.14), new Double(11.11) });
    f(new Object[] {"un", "dos", "tres" });
    f(new Object[] {new A(), new A(), new A()});
  }
} ///:~
```

En este punto, no hay mucho que pueda hacerse con estos objetos desconocidos, y el programa usa la conversión automática **String** para hacer algo útil con cada **Object**. En el Capítulo 12, que cubre la *identificación de tipos en tiempo de ejecución* (*Run-time type identification*, RTTI), se aprenderá a descubrir el tipo exacto de objetos así, de forma que se pueda hacer algo más interesante con ellos.

Arrays multidimensionales

Java permite crear fácilmente arrays multidimensionales:

```
//: c04:ArrayMultidimensional.java
// Creando arrays multidimensionales.
import java.util.*;

public class ArrayMultidimensional {
  static Random aleatorio = new Random();
  static int pAleatorio(int modulo) {
    return Math.abs(aleatorio.nextInt()) % modulo + 1;
  }
  static void visualizar(String s) {
    System.out.println(s);
  }
  public static void main(String[] args) {
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    for(int i = 0; i < a1.length; i++)
      for(int j = 0; j < a1[i].length; j++)
        visualizar("a1[" + i + "][" + j +
            "] = " + a1[i][j]);
    // array 3-D de longitud fija:
    int[][][] a2 = new int[2][2][4];
    for(int i = 0; i < a2.length; i++)
      for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length;
            k++)
          visualizar("a2[" + i + "][" +
              j + "][" + k +
              "] = " + a2[i][j][k]);
    // array 3-D con vectores de longitud variable:
    int[][][] a3 = new int[pAleatorio(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pAleatorio(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pAleatorio(5)];
    }
    for(int i = 0; i < a3.length; i++)
      for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length;
            k++)
```

```
            visualizar("a3[" + i + "][" +
                j + "][" + k +
                "] = " + a3[i][j][k]);
    // Array de objetos no primitivos:
    Integer[][] a4 = {
      { new Integer(1), new Integer(2)},
      { new Integer(3), new Integer(4)},
      { new Integer(5), new Integer(6)},
    };
    for(int i = 0; i < a4.length; i++)
      for(int j = 0; j < a4[i].length; j++)
        visualizar("a4[" + i + "][" + j +
            "] = " + a4[i][j]);
    Integer[][] a5;
    a5 = new Integer[3][];
    for(int i = 0; i < a5.length; i++) {
      a5[i] = new Integer[3];
      for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
    }
    for(int i = 0; i < a5.length; i++)
      for(int j = 0; j < a5[i].length; j++)
        visualizar("a5[" + i + "][" + j +
            "] = " + a5[i][j]);
  }
} ///:~
```

El código utilizado para imprimir utiliza el método **length**, de forma que no depende de tamaños fijos de array.

El primer ejemplo muestra un array multidimensional de tipos primitivos. Se puede delimitar cada vector del array por llaves:

```
int[][] a1 = {
        { 1, 2, 3, },
        { 4, 5, 6, },
    };
```

Cada conjunto de corchetes nos introduce en el siguiente nivel del array.

El segundo ejemplo muestra un array de tres dimensiones asignado con **new**. Aquí, se asigna de una sola vez todo el array:

```
int[][][] a2 = new int[2][2][4];
```

Pero el tercer ejemplo muestra que cada vector en los arrays que conforman la matriz pueden ser de cualquier longitud:

```
int[][][] a3 = new int[pAleatorio(7)][][];
     for(int i = 0; i < a3.length; i++) {
       a3[i] = new int[pAleatorio(5)][];
       for(int j = 0; j < a3[i].length; j++)
         a3[i][j] = new int[pAleatorio(5)];
     }
```

El primer **new** crea un array con un primer elemento de longitud aleatoria, y el resto, indeterminados. El segundo **new** de dentro del bucle **for** rellena los elementos pero deja el tercer índice indeterminado hasta que se acometa el tercer **new**.

Se verá en la salida que los valores del array que se inicializan automáticamente a cero si no se les da un valor de inicialización explícito.

Se puede tratar con arrays de objetos no primitivos de forma similar, lo que se muestra en el cuarto ejemplo, que demuestra la habilidad de englobar muchas expresiones **new** entre llaves:

```
Integer[][] a4 = {
       { new Integer(1), new Integer(2)},
       { new Integer(3), new Integer(4)},
       { new Integer(5), new Integer(6)},
     };
```

El quinto ejemplo muestra cómo se puede construir pieza a pieza un array de objetos no primitivos:

```
Integer[][] a5;
     a5 = new Integer[3][];
     for(int i = 0; i < a5.length; i++) {
       a5[i] = new Integer[3];
       for(int j = 0; j < a5[i].length; j++)
         a5[i][j] = new Integer(i*j);
     }
```

El **i*j** es simplemente para poner algún valor interesante en el **Integer**.

Resumen

El constructor, mecanismo aparentemente elaborado de inicialización, proporciona un importante mecanismo para realizar la inicialización. Cuando Stroustrup estaba diseñando C++, una de las primeras observaciones que hizo sobre la productividad de C era relativa a la inicialización de las variables erróneas que causan un porcentaje significativo de los problemas de programación. Estos tipos de fallos son difíciles de encontrar, y hay aspectos similares que pueden aplicarse a la limpieza errónea. Dado que los constructores permiten *garantizar* la inicialización correcta y la limpieza (el compilador no permitirá que un objeto se cree sin los constructores pertinentes), se logra un control y seguridad completos.

En C++, la destrucción es bastante importante porque los objetos creados con **new** deben ser destruidos explícitamente. En Java, el recolector de basura libera automáticamente la memoria de todos

los objetos, por lo que el método de limpieza equivalente es innecesario en Java en la mayoría de ocasiones. En los casos en los que no es necesario un comportamiento al estilo de un destructor, el recolector de basura de Java simplifica enormemente la programación, y añade un elevado y necesario nivel de seguridad a la gestión de memoria. Algunos recolectores de basura pueden incluso limpiar otros recursos como los gráficos y los manejadores de ficheros. Sin embargo, el recolector de basura añade un coste en tiempo de ejecución, cuyo gasto es difícil de juzgar, debido a la lentitud de los intérpretes de Java existentes en el momento de escribir el presente libro. Cuando cambie esto, se podrá descubrir si la sobrecarga del recolector de basura excluirá el uso de Java para determinados tipos de programas. (Uno de los aspectos es la falta de predicción del recolector de basura.)

Dado que se garantiza la construcción de todos los objetos, de hecho, hay más aspectos que los aquí descritos. En particular, al crear nuevas clases usando la *agregación* o la *herencia* también se mantiene la garantía de construcción, aunque es necesaria cierta sintaxis para dar soporte a esto. Se aprenderá todo lo relativo a la agregación, la herencia y cómo afectan éstas operaciones a los constructores en los capítulos siguientes.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com.*

1. Crear una clase con el constructor por defecto (el que no tiene parámetros) que imprima un mensaje. Crear un objeto de esta clase.

2. Añadir un constructor sobrecargado al Ejercicio 1, que tome un **String** como parámetro y lo imprima junto con el mensaje.

3. Crear un array de referencias a objetos de la clase creada en el Ejercicio 2, pero no crear los objetos a asignar al array. Al ejecutar el programa, tomar nota de si se imprimen los mensajes de inicialización del constructor.

4. Completar el Ejercicio 3 creando los objetos a asociar al array de referencias.

5. Crear un array de objetos **String** y asignar una cadena de caracteres a cada elemento. Imprimir el array utilizando un bucle **for**.

6. Crear una clase **Perro** con un método **ladrar( )** sobrecargado. Este método debería sobrecargarse en base a varios tipos de datos primitivos, e imprimir distintos tipos de ladridos, aullidos, etc. dependiendo de la versión sobrecargada que se invoque. Escribir un método **main( )** que llame a todas las distintas versiones.

7. Modificar el Ejercicio 6 de forma que dos de los métodos sobrecargados tengan dos argumentos (de dos tipos distintos), pero en orden inverso entre sí. Verificar que funciona.

8. Crear una clase sin constructor, y crear un objeto de esa clase en **main( )** para verificar que el constructor por defecto se invoca automáticamente.

9. Crear una clase con dos métodos. Dentro del primer método, invocar al segundo dos veces: la primera vez sin utilizar **this**, y la segunda, usando **this**.

10. Crear una clase con dos constructores (sobrecargados). Utilizando **this**, invocar al segundo constructor dentro del primero.

11. Crear una clase con un método **finalize( )** que imprima un mensaje. En **main( )**, crear un objeto de esa clase. Explicar el funcionamiento del programa.

12. Modificar el Ejercicio 11 de forma que siempre se llame a **finalize( )**.

13. Crear una clase llamada **Tanque** que pueda rellenarse y vaciarse, y que tenga una *condición de muerte* que tenga que estar vacía al eliminar el objeto. Escribir un **finalize( )** que verifique esta condición de muerte. En el método **main( )**, probar los escenarios posibles que puedan ocurrir al usar **Tanque**.

14. Crear una clase que contenga un **int** y un **char** no inicializados, e imprimir sus valores para verificar que Java realiza la inicialización por defecto.

15. Crear una clase que contenga una referencia **String** sin inicializar. Demostrar que Java inicializa esta referencia a **null**.

16. Crear una clase con un campo **String** que se inicialice en el momento de la definición y otra que inicialice el constructor. ¿Cuál es la diferencia entre los dos enfoques?

17. Crear una clase con un campo **estático String** que se inicialice en el momento de la definición, y otra que sea inicializada por un bloque **estático**. Añadir un método **estático** que imprima ambos campos y demuestre que ambos se inicializan antes de usarse.

18. Crear una clase con un **String** que se inicialice usando "inicialización de instancias". Describir un uso de esa característica (una descripción distinta de la que se especifica en este libro).

19. Escribir un método que cree e inicialice un array bidimensional de datos de tipo **double**. El tamaño del array vendrá determinado por los parámetros del método, y los valores de inicialización vendrán determinados por un rango delimitado por sus valores superior e inferior, parámetros ambos también del método. Crear un segundo método que imprima el array generado por el primer método. En el método **main( )** probar los métodos creando e imprimiendo varios arrays de distintos tamaños.

20. Repetir el Ejercicio 19 para un array tridimensional.

21. Comentar la línea marcada (1) en **StaticExplicito.java** y verificar que la cláusula de inicialización estática no es invocada. Ahora, quitar la marca de comentario de alguna de las líneas marcadas (2) y verificar que se invoca a la cláusula de inicialización estática. Ahora quitar la marca de comentario de la otra línea marcada (2) y verificar que la inicialización estática sólo se da una vez.

22. Experimentar con **Basura.java** ejecutando el programa utilizando los argumentos "rec", "finalizar" o "todo". Repetir el proceso y ver si detecta patrones en la salida. Cambiar el código de forma que se llame a **System.runFinalization( )** *antes* que a **System.gc( )** y observar los resultados.

5: Ocultar la implementación

Una consideración primordial del diseño orientado a objetos es "la separación de aquellas cosas que varían de aquéllas que permanecen constantes".

Esto es especialmente importante en el caso de las bibliotecas. El usuario (el *programador cliente*) de la biblioteca debe ser capaz de confiar en la parte que usa, y saber que no necesita reescribir el código si se lanza una nueva versión de esa biblioteca. Por otro lado, el creador de la biblioteca debe tener la libertad para hacer modificaciones y mejoras con la certeza de que el código del programador cliente no se verá afectado por estos cambios.

Esto puede lograse mediante una convención. Por ejemplo, el programador de la biblioteca debe acordar no eliminar métodos existentes al modificar una clase de la biblioteca, dado que eso destruiría el código del programador cliente. El caso contrario sería más problemático. En el caso de un atributo ¿cómo puede el creador de la biblioteca saber qué atributos son los que los programadores clientes han usado? Esto también ocurre con aquellos métodos que sólo son parte de la implementación de la clase, pero que no se diseñaron para ser usados directamente por el programador cliente. Pero, ¿qué ocurre si el creador de la biblioteca desea desechar una implementación antigua y poner una nueva? El cambio de cualquiera de esos miembros podría romper el código de un programador cliente. Por consiguiente, el creador de la biblioteca se encuentra limitado, y no puede cambiar nada.

Para solucionar este problema, Java proporciona *especificadores de acceso* para permitir al creador de la biblioteca decir qué está disponible para el programador cliente, y qué no. Los niveles de control de acceso desde el "acceso máximo" hasta el "acceso mínimo" son **public**, **protected**, "friendly" (para el que no existe palabra clave), y **private**. Por el párrafo anterior podría pensarse que, al igual que el diseñador de la biblioteca, se deseará mantener "private" tanto como sea posible, y exponer únicamente los métodos que se desee que use el programador cliente. Esto es completamente correcto, incluso aunque frecuentemente no es intuitivo para aquéllos que programan en otros lenguajes (especialmente C), y se utilizan para acceder a todo sin restricciones. Para cuando acabe este capítulo, el lector debería convencerse del valor del control de accesos en Java.

Sin embargo, el concepto de una biblioteca de componentes y el control sobre quién puede acceder a los componentes de esa biblioteca están completos. Todavía queda la cuestión de cómo se empaquetan los componentes para formar una unidad cohesiva. Esto se controla en Java con la palabra clave **package**, y los especificadores de acceso se ven en la medida en que una clase se encuentre en un mismo o distinto paquete. Por tanto, para empezar este capítulo, se aprenderá cómo ubicar los componentes de las bibliotecas en paquetes. Posteriormente, uno será capaz de comprender el significado completo de los especificadores de acceso.

El paquete: la unidad de biblioteca

Un paquete es lo que se obtiene al utilizar la palabra clave **import** para importar una biblioteca completa, como en:

```
import java.util.*;
```

Esto trae la biblioteca de utilidades entera, que es parte de la distribución estándar de Java. Dado que, por ejemplo, la clase **ArrayList** se encuentra en **java.util** es posible especificar el nombre completo **java.util. ArrayList** (lo cual se puede hacer sin la sentencia **import**) o bien se puede simplemente decir **ArrayList** (gracias a la sentencia **import**).

Si se desea incorporar una única clase, es posible nombrarla sin la sentencia **import**:

```
import java.util.ArrayList;
```

Ahora es posible hacer uso de **ArrayList**, aunque no estarán disponibles ninguna de las otras clases de **java.util**.

La razón de todas estas importaciones es proporcionar un mecanismo para gestionar los "espacios de nombres". Los nombres de todas las clases miembros están aislados unos de otros. Un método **f( )** contenido en la clase **A** no colisionará con un método **f( )** que tiene la misma lista de argumentos, dentro de la clase **B**. Pero, ¿qué ocurre con los nombres de clases? Supóngase que se crea una clase **Pila** que se instala en una máquina que ya tiene una clase **Pila** escrita por otra persona. Con Java en Internet, esto podría incluso ocurrir sin que el usuario lo sepa, dado que es posible que algunas clases se descarguen automáticamente en el proceso de ejecutar un programa Java.

Esta potencial colisión de nombres justifica la necesidad de tener control sobre los espacios de nombre en Java, y de tener la capacidad de crear un nombre completamente único sin que importen las limitaciones de Internet.

Hasta ahora, la mayoría de los ejemplos de este libro se incluían en un único fichero y están diseñados para un uso local, por lo que no han tenido que hacer uso de los nombres de paquetes. (En este caso el nombre de clase se ubica en el "paquete por defecto".) Ésta es ciertamente una opción, y con motivo de mantener la máxima simplicidad, se usará este enfoque siempre que sea posible en todo el resto del libro. Sin embargo, si se planifica crear bibliotecas o programas que se relacionan con otros programas Java de la misma máquina, hay que pensar en evitar las colisiones entre nombres de clases.

Cuando se crea un fichero de código fuente en Java, se crea lo que comúnmente se denomina una *unidad de compilación* (en ocasiones se denomina una *unidad de traducción*). Cada una de estas unidades tiene un nombre que acaba en **.java**, y dentro de la unidad de compilación puede haber una única clase **pública**, sino, el compilador se quejará. El resto de clases de esa unidad de compilación, si es que hay alguna, quedan ocultas para todo lo exterior al paquete al *no* ser **pública**, y constituyen clases de "apoyo" para la clase **pública** principal.

Cuando se compila un fichero **.java**, se obtiene un fichero de salida que tiene exactamente el mismo nombre pero tiene extensión **.class** *por cada clase* del fichero **.java**. Por tanto, se puede acabar teniendo bastantes ficheros **.class** partiendo de un número pequeño de ficheros **.java**. Si se programa haciendo uso de un lenguaje compilado, puede que uno esté acostumbrado a que el compilador devuelva un fichero en un formato intermedio (generalmente un fichero "obj") que se empaqueta junto con otros de su misma clase utilizando, bien un montador (para crear un fichero ejecutable) o una biblioteca. Java no funciona así. Un programa en acción es un compendio de ficheros **.java**, que pueden empaquetarse y comprimirse en un fichero JAR (utilizando la herramienta **jar** de Java).

El intérprete de Java es el responsable de encontrar, cargar e interpretar estos ficheros[1].

Una biblioteca también es un conjunto de estos ficheros de clase. Cada fichero tiene una clase que es **pública** (no es obligatorio introducir una clase **pública**, pero lo habitual es hacerlo así), de forma que hay un componente por cada fichero. Si se desea indicar que todos estos componentes (que se encuentran en sus propios ficheros separados **.java** y **.class**) permanezcan unidos, es necesaria la intervención de la palabra clave **package**.

Cuando se dice:

```
package mipaquete;
```

(al principio de un archivo), si se usa la sentencia **package**, ésta *debe* aparecer en la primera línea que no sea un comentario del fichero), se está indicando que esa unidad de compilación es parte de una biblioteca de nombre **mipaquete**. O, dicho de otra forma, se está diciendo que el nombre de la clase **pública** incluida en esa unidad de compilación se encuentra bajo el paraguas del nombre, **mipaquete**, y si alguien quiere utilizar el nombre, deben, o bien especificar completamente el nombre o bien usar la palabra clave **import** en combinación con **mipaquete** (utilizando las opciones descritas previamente). Fíjese que la convención para los nombres de paquete de Java dice que se usen únicamente letras minúsculas, incluso cuando hay más de una palabra.

Por ejemplo, supóngase que el nombre del fichero es **MiClase.java**. Esto significa que puede haber una y sólo una clase **pública** en ese fichero, y el nombre de esa clase debe ser **MiClase** (incluidas las mayúsculas y minúsculas):

```
package mipaquete;
public class MiClase {
  // . . .
```

Ahora, si alguien desea usar **MiClase** o, por cualquier motivo, cualquiera de las clases **públicas** de **mipaquete**, debe usar la palabra clave **import** para lograr que estén disponibles el/los nombres de **mipaquete**. La alternativa es dar el nombre completo:

```
mipaquete.MiClase m = new mipaquete.MiClase();
```

[1] No hay nada en Java que obligue al uso de un intérprete. Existen compiladores de código nativo Java que generan un único fichero ejecutable.

La palabra clave **import** puede lograr lo mismo pero de manera bastante más clara:

```
import mipaquete
// . . .
MiClase m = MiClase();
```

Merece la pena recordar que lo que las palabras clave **package** e **import** permiten hacer, como diseñador de bibliotecas, es dividir el espacio de nombres único y global, de forma que no se tengan colisiones de nombres, sin que importe cuánta gente se conecte a Internet y empiece a escribir clases en Java.

Creando nombres de paquete únicos

Podría observarse que, dado que un paquete nunca se llega a "empaquetar" en un fichero único, un mismo fichero podría estar constituidc por muchos ficheros **.class**, y esto podría ser fuente de desorden y confusión. Para evitarlo, algo lógico es ubicar todos los ficheros **.java** de un paquete particular en un mismo directorio; es decir, hacer uso de la estructura de ficheros jerárquica del sistema operativo y sacar provecho de ella. Ésta es una de las maneras en que Java referencia el problema del desorden; se verá otra manera después, cuando se presente la utilidad **jar**.

La agrupación de los ficheros de paquete en un único subdirectorio soluciona otros dos problemas: la creación de nombres de paquete únicos, y la localización de esas clases que podrían estar enterradas en algún lugar de la estructura de directorios. Esto se logra, tal y como se presentó en el Capítulo 2, codificando camino de localización del fichero **.class** en el nombre del **paquete**. El compilador obliga a que esto sea así, pero por convención, la primera parte del nombre de un **paquete** es el nombre del dominio Internet del creador de la clase, eso sí, dado la vuelta. Dado que está garantizado que los nombres de dominio de Internet sean únicos, *si* se sigue esta convención se garantiza que el nombre del **paquete** sea único y, por consiguiente, nunca habrá colisiones de nombres (es decir, hasta que se pierde el nombre de dominio y alguien se hace con él y empieza a escribir código Java con los mismos nombres de ruta con los que lo hizo). Por supuesto, si se dispone de un nombre de dominio propio, es necesario fabricar en primer lugar una combinación única (como, por ejemplo, la formada por el nombre y apellidos) para crear nombres de paquete únicos. Si se ha decidido comenzar a publicar código Java merece la pena el esfuerzo, relativamente pequeño, de conseguir en primer lugar un nombre de dominio.

La segunda parte de este truco es la resolución del nombre de **paquete** en un directorio de la máquina, de forma que cuando se ejecuta un programa Java y necesita cargar el fichero **.class** (lo que ocurre dinámicamente, en el punto en el que el programa necesite crear un objeto de esa clase en particular, o la primera vez que se accede a un miembro **estático** de la clase), pueda localizar el directorio en el que reside el fichero **.class**.

El intérprete de Java procede de la siguiente forma. En primer lugar, encuentra la variable de entorno CLASSPATH (establecida mediante el sistema operativo, a veces por parte del programa de instalación de Java, o una herramienta basada en Java de la propia máquina). CLASSPATH contiene uno o más directorios utilizados como raíz para la búsqueda de ficheros **.class**. A partir de esa raíz, el intérprete toma el nombre de paquete y reemplaza cada punto por una barra para generar un

nombre relativo a la raíz CLASSPATH (de forma que el **paquete foo.bar.baz** se convierte en **foo\bar\baz** o en **foo/bar/baz** en función del sistema operativo instalado). A continuación, se concatena este nombre con las distintas entradas de la variable CLASSPATH. Es en este momento cuando se busca por el archivo **.class** que coincida en nombre con la clase que se está intentando crear (también busca algunos directorios estándares relativos al directorio en el que reside el intérprete Java).

Para entenderlo, considérese mi nombre de dominio, que es **bruceeckel.com**. Dando la vuelta a esto, **com.bruceeckel** establece el único nombre global a utilizar en todas mis clases. (La extensión com, edu, org, etc., se ponía en mayúsculas en las primeras versiones de los paquetes Java, pero esto se ha cambiado en Java 2, de forma que todo el nombre de paquete se escribe en minúsculas.) Posteriormente se puede subdividir este nombre diciendo que se quiere crear una biblioteca llamada **simple**, por lo que acabaremos con un nombre de paquete:

```
package com.bruceeckel.simple;
```

Ahora, este nombre de paquete puede usarse como un espacio de nombre paraguas para los siguientes dos archivos:

```
//: com:bruceeckel:simple:Vector.java
// Creando un paquete.
package com.bruceeckel.simple;

public class Vector {
  public Vector() {
    System.out.println(
      "com.bruceeckel.util.Vector");
  }
} ///:~
```

Cuando uno crea sus propios paquetes, se descubre que la sentencia **package** debe ser la primera del archivo de código que no sea un comentario dentro del archivo. El segundo archivo es muy parecido:

```
//: com:bruceeckel:simple:Lista.java
// Creando un paquete.
package com.bruceeckel.simple;

public class Lista {
  public Lista() {
    System.out.println(
      "com.bruceeckel.util.Lista");
  }
} ///:~
```

Ambos ficheros se encuentran ubicados en el subdirectorio:

```
C:\DOC\JavaT\com\bruceeckel\simple
```

Si se empieza a recorrer esta trayectoria se puede componer el nombre de paquete **com.bruceeckel.simple**, pero ¿qué ocurre con la primera parte de la trayectoria? De esto se encarga la variable de entorno CLASSPATH:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Puede verse que la variable CLASSPATH puede contener más de un directorio de búsqueda, todos ellos alternativos.

Sin embargo, hay una variación cuando se usan archivos JAR. Se debe poner el nombre del archivo JAR en la trayectoria de clases CLASSPATH, no sólo la trayectoria en la que se encuentra. Así, para un JAR de nombre **uva.jar**, esta variable será:

```
CLASSPATH=.;D:\JAVA\LIB;C:\sabores\uva.jar
```

Una vez que se ha establecido correctamente el valor de esta variable, buscará el archivo en cualquiera de sus directorios:

```
//: c05:PruebaBiblioteca.java
// Utiliza la biblioteca.
import com.bruceeckel.simple.*;

public class PruebaBiblioteca {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} ///:~
```

Cuando el compilador encuentra la sentencia **import**, empieza a buscar en los directorios especificados por CLASSPATH, buscando el subdirectorio com\bruceeckel\simple, y buscando después los ficheros compilados de nombres adecuados (**Vector.class** para **Vector** y **List.class** para **List**). (Fíjese que, tanto las clases, como los métodos deseados de **Vector** y **List,** deben ser **públicos**).

Establecer la variable CLASSPATH era tan problemático para los usuarios de Java principiantes (como lo era para mí cuando empecé) que Sun ha hecho el JDK de Java 2 algo más inteligente. Se descubrirá que, al instalarlo, incluso si no se establece un CLASSPATH, se podrán compilar y ejecutar programas básicos de Java. Para compilar y ejecutar el paquete código de este libro (disponible en el CD ROM empaquetado junto con este libro, o en *www. BruceEckel.com*), sin embargo, se necesitará hacer algunas modificaciones al CLASSPATH (éstas se explican en el paquete de código fuente).

Colisiones

¿Qué ocurre si se importan dos bibliotecas vía * que incluyen los mismos nombres? Por ejemplo, supóngase que un programa hace:

```
import com.bruceeckel.simple.*;
import java.util.*;
```

Dado que **java.util.*** también contiene una clase **Vector**, esto causa una colisión potencial. Sin embargo, mientras no se escriba el código que, de hecho, cause la colisión, todo va bien —esto es bueno porque de otra forma, uno podría acabar tecleando multitud de código para evitar colisiones que nunca ocurrirían.

La colisión *ocurre* si ahora se intenta crear un **Vector**:

```
Vector v = new Vector();
```

¿A qué **Vector** se refiere? El compilador no puede saberlo, y tampoco puede el lector. Por tanto el compilador se queja y obliga a especificar. Si se desea el **Vector** estándar de Java, por ejemplo, hay que decir:

```
java.util.Vector v = new java.util.Vector();
```

Dado que esto (junto con la variable CLASSPATH) especifica completamente la localización de ese **Vector**, no hay necesidad de la sentencia **import.java.util.***, a menos que se esté utilizando algo más de **java.util**.

Una biblioteca de herramientas a medida

Con estos conocimientos, ahora cada uno puede crear sus propias bibliotecas de herramientas para reducir o eliminar el código duplicado. Considérese, por ejemplo, que se está creando un alias para **System.out.println( )** para reducir el código a teclear. Éste podría ser parte de un paquete llamado **herramientas**:

```
//: com:bruceeckel:herramientas:P.java
// El atajo P.rint y P.rintln.
package com.bruceeckel.herramientas;

public class P {
  public static void rint(String s){
    System.out.print(s);
  }
  public static void rintln(String s) {
    System.out.println(s);
  }
} ///:~
```

Se puede usar este atajo para usar un **String**, bien con un retorno de carro al final (**P.rintln( )**) o sin él (**P.rint( )**).

Se puede adivinar que este archivo debe estar ubicado en un directorio de los especificados en CLASSPATH, y que continúe por **com/bruceeckel/herramientas**. Una vez compilado, el fichero **P.class** puede usarse en cualquier lugar del sistema con una sentencia **import**:

```
//: c05:PruebaHerramienta.java
// Utiliza la biblioteca herramientas.
```

```
import com.bruceeckel.herramientas.*;

public class PruebaHerramienta {
  public static void main(String[] args) {
    P.rintln("¡Disponible de ahora en adelante!");
    P.rintln("" + 100); // Obligar a que sea un String
    P.rintln("" + 100L);
    P.rintln("" + 3.14159);
  }
} ///:~
```

Obsérvese que se puede forzar a cualquier objeto a transformarse en una representación en forma de **String**, poniéndolos en una expresión **String**; en el caso anterior, se hace uso de un truco: comenzar la expresión con un **String** vacío. Pero esto recuerda una observación interesante. Si se invoca a **System.out.println(100)**, funciona sin tener que convertirlo a **String**. Con algo de sobrecarga, se puede conseguir que la clase **P** haga también esto (planteado como ejercicio al final del presente capítulo).

Por tanto, de ahora en adelante, cuando construya una nueva utilidad, se puede añadir al directorio **herramientas**. (O al directorio **util** o **herramientas** de cada uno.)

Utilizar el comando import para cambiar el comportamiento

Una característica que Java no ha heredado de C es la *compilación condicional*, que permite modificar un switch y obtener distintos comportamientos sin necesidad de variar ninguna otra parte del código. La razón por la que esta característica no se incluyó en Java es probablemente el hecho de que se utiliza en C fundamentalmente para resolver problemas de multiplataforma: se compilan distintas porciones de código en función de la plataforma para la que se está compilando cada código. Puesto que se pretende que Java sea multiplataforma automáticamente, una característica así no es necesaria.

Sin embargo, hay otros usos de gran valor en la compilación condicional. Un uso muy común es la depuración de código. Los aspectos de depuración se habilitan durante el desarrollo, y se deshabilitan en el lanzamiento del producto. A Allen Holub (*www.holub.com*) se le ocurrió la idea de utilizar paquetes para simular la compilación condicional. Hizo uso de esta idea para crear una versión Java del *mecanismo de afirmaciones* —tan útil en C—, mediante el que se puede decir "esto debería ser verdad" o "esto debería ser falso" y si la sentencia no está de acuerdo con el afirmación, ya se averiguará. Este tipo de herramienta supone una gran ayuda durante la fase de depuración.

He aquí la clase que se utilizará para depuración:

```
//: com:bruceeckel:herramientas:depurar:Afirmacion.java
// Herramienta de aserto para la depuración;
package com.bruceeckel.tools.debug;
```

```
public class Afirmacion {
  private static void error(String msg) {
    System.err.println(msg);
  }
  public final static void es_cierto(boolean exp) {
    if(!exp) error("Fallo la afirmacion");
  }
  public final static void es_falso(boolean exp) {
    if(exp) error("Fallo la afirmacion");
  }
  public final static void
        es_cierto(boolean exp, String mensaje){
    if (!exp) error("Fallo la afirmacion: " + mensaje);
  }
  public final static void
  es_falso(boolean exp, String msg) {
    if(exp) error("Fallo la afirmacion: " + mensaje);
  }
} ///:~
```

Esta clase simplemente encapsula pruebas de valores lógicos, que imprimen mensajes de error si fallan. En el Capítulo 10, se conocerá una herramienta más sofisticada para tratar con errores, denominada *manejo de excepciones*, pero el método **error( )** será suficiente mientras tanto.

La salida se imprime en el "flujo de datos" de la consola de *error estándar* escribiendo en **System.err**.

Cuando se desee hacer uso de esta clase, basta con añadir en el programa la línea:

```
import com.bruceeckel.herramientas.depurar.*;
```

Para retirar las afirmaciones que pueda lanzar el código, se crea una segunda clase **Afirmacion**, pero en un paquete distinto:

```
//: com:bruceeckel:herramientas:depurar:Afirmacion.java
// Desactivar la salida de la afirmacion
// de forma que se pueda lanzar el programa.
Package com.bruceeckel.herramientas;

public class Afirmacion {
  public final static void es_cierto(boolean exp) {}
  public final static void es_falso(boolean exp) {}
  public final static void
  es_cierto(boolean exp, String mensaje) {}
  public final static void
  es_falso(boolean exp, String mensaje) {}
} //:~
```

Ahora, si se cambia la sentencia **import** anterior a:

```
import com.bruceeckel.herramientas.*;
```

El programa dejará de imprimir afirmaciones. He aquí un ejemplo:

```
//: c05:PruebaAfirmacion.java
// Demostrando la herramienta de afirmación.
// Comentar y quitar el comentario
// de la línea siguiente para cambiar
// el comportamiento del aserto:
import com.herramientas.depurar.debug.*;
// import com.bruceeckel.herramientas.*;

public class PruebaAfirmacion {
   public static void main(String[] args) {
      afirmacion.es_cierto((2 + 2) == 5);
      afirmacion.es_falso((1 + 1) == 2);
      afirmacion.es_cierto((2 + 2) == 5, "2 + 2 == 5");
      afirmacion.es_falso((1 + 1) == 2, "1 +1 != 2");
   }
} ///:~
```

Al cambiar el **paquete** que se importa, se cambia el código de la versión en depuración a la versión de producción. Esta técnica puede usarse para cualquier tipo de valor condicional.

Advertencia relativa al uso de paquetes

Merece la pena recordar que cada vez que se cree un paquete, implícitamente se está especificando una estructura de directorios al dar un nombre a un paquete. El paquete *debe* residir en el directorio indicado por su nombre, que debe ser un directorio localizable a partir de CLASSPATH.

Experimentar con la palabra clave **package**, puede ser un poco frustrante al principio, puesto que, a menos que se adhiera al nombre del paquete la regla de trayectorias de directorios, se obtendrán numerosos mensajes en tiempo de ejecución que indican que no es posible localizar una clase en particular, incluso si esa clase reside en ese mismo directorio. Si se obtiene uno de estos mensajes, debe tratar de modificar la sentencia *package*, y cuando funcione se sabrá dónde residía el problema.

Modificadores de acceso en Java

Al utilizarlos, los modificadores de acceso **public**, **protected** y **private** se ubican delante de cada definición de cada miembro de la clase, sea un atributo o un método. Cada modificador de acceso controla el acceso sólo para esa definición en particular. Éste es diferente a C++, lenguaje en el que

el controlador de acceso controla todas las definiciones que lo sigan hasta la aparición del siguiente modificador de acceso.

De una manera u otra, todo tiene asignado algún tipo de modificador de acceso. En las secciones siguientes, se aprenderán los distintos tipos de accesos, comenzando por el acceso por defecto.

"Amistoso" ("Friendly")

¿Qué ocurre si no se indica ningun tipo de especificador de acceso, como en todos los ejemplos anteriores de este capítulo? El acceso por defecto no tiene ninguna palabra clave asociada, pero generalmente se le denomina acceso "amistoso". Significa que todas las demás clases del paquete actual tienen acceso al miembro amistoso, pero de cara a todas las clases de fuera del paquete, el miembro aparenta ser **privado**. Dado que una unidad de compilación —un fichero— puede pertenecer sólo a un único paquete, todas las clases de una única unidad de compilación son automáticamente "Amistosas" entre sí. Por consiguiente, se dice que los elementos "Amistosos" tienen *acceso paquete*.

El acceso amistoso permite agrupar clases relacionadas en un mismo paquete de forma que éstas puedan interactuar entre sí de manera sencilla. Al poner clases juntas en un paquete (garantizando por consiguiente el acceso mutuo "Amistoso" a sus miembros; por ejemplo, marcándolos como amigos) se "posee" el código de ese paquete. Tiene sentido que el único código que se posee debería tener acceso amistoso al resto de código propio. Podría decirse que el acceso amistoso da un significado o razón para agrupar juntas las clases de un paquete. En muchos lenguajes, la forma de organizar las definiciones de los ficheros puede ser obligatoria, pero en Java obliga a que cada uno las organice de manera sensata. Además, probablemente se excluirán las clases que no deberían tener acceso a las clases que están siendo definidas en el paquete actual.

La clase controla qué código tiene acceso a sus miembros. No hay ningún truco para "irrumpir" en ella. No se puede mostrar el código de otros paquetes y decir: "¡Hola, soy un amigo de **Bob**!", y esperar que se vean los miembros **protegidos**, "amistosos", y **privados** de **Bob**. La única manera de garantizar los accesos a un miembro es:

1. Hacer el miembro **público**. Posteriormente, todo el mundo, en todas partes, podría acceder a él.

2. Hacer el miembro amistoso no indicando ningún especificador de acceso, y poner las otras clases en el mismo paquete. Así, las otras clases pueden acceder al miembro.

3. Como se verá en el Capítulo 6, cuando se presente la herencia, una clase heredada puede acceder a un miembro **protegido** al igual que a un miembro **público** (pero no a los miembros **privados**). Puede acceder a los miembros "amistosos" sólo si las dos clases se encuentran en el mismo paquete. Pero no hay que preocuparse de esto ahora.

4. Proporcionar métodos "obtener/establecer" *("get/set")* que lean y cambien el valor. Éste es el enfoque más habitual en términos de POO, y es fundamental para los JavaBeans como se verá en el Capítulo 13.

public: acceso a interfaces

Cuando se usa la palabra clave **public**, significa que la declaración de miembro que continúe inmediatamente a **public** estará disponible a todo el mundo, y en especial al programador cliente que hace uso de la biblioteca. Supóngase que se define un paquete **postre**, que contiene la siguiente unidad de compilación:

```
//: c05:postre:Galleta.java
// Crea una biblioteca.
package c05.postre;

public class Galleta {
  public Galleta() {
   System.out.println("Constructor de Galleta");
  }
  void morder() { System.out.println("morder"); }
} ///:~
```

Recuérdese que **Galleta.java** debe residir en un subdirectorio de nombre **postre**, en un directorio bajo **c05** (que se corresponde con el Capítulo 5 de este libro) que debe estar bajo uno de los directorios de CLASSPATH. No hay que cometer el error de pensar que Java siempre buscará en el directorio actual como uno de los directorios de partida para la búsqueda. Si no se tiene un '.' como una de las rutas del CLASSPATH, Java no buscará ahí.

Ahora, si se crea un programa que haga uso de **Galleta**:

```
//: c05:Cena.java
// Hace uso de la biblioteca.
import c05.postre.*;

public class Cena {
  public Cena() {
   System.out.println("Constructor cena");
  }
  public static void main(String[] args) {
    Galleta x = new Galleta();
    //! x.morder(); // No se puede acceder
  }
} ///:~
```

se puede crear un objeto **Galleta**, dado que su constructor es **público** y la clase es **pública**. (Se profundizará más tarde en el concepto de **público**.) Sin embargo, el método **morder( )** es inaccesible dentro de **Cena.java** puesto que **morder( )** es amistoso sólo dentro del paquete **postre**.

El paquete por defecto

Uno podría sorprenderse de descubrir que el código siguiente compila, incluso aunque aparenta transgredir las reglas:

```
//: c05:Tarta.java
// Accede a una clase de una
// unidad de compilación distinta.

class Tarta {
  public static void main(String[] args) {
    Pastel x = new Pastel();
    x.f();
  }
} ///:~
```

En un segundo archivo del mismo directorio:

```
//: c05:Pastel.java
// La otra clase.

class Pastel {
  void f() { System.out.println("Pastel.f()"); }
} ///:~
```

Inicialmente uno podría pensar que se trata de archivos completamente independientes, y sin embargo, **Tarta** es incluso capaz de crear un objeto **Pastel**, e invocar a su método **f( )**. (Fíjese que debe tenerse el '.' en CLASSPATH para que los archivos se compilen.) Normalmente se pensaría que **Pastel** y **f( )** son amistosos y por consiguiente, no están disponibles para **Tarta**. *Son* amistosos —hasta ahí es correcto. La razón por la que están disponibles en **Tarta.java** es que se encuentran en el mismo directorio y no tienen ningún nombre de paquete explícito. Java trata a los archivos así como si fueran parte implícita del "paquete por defecto" de ese directorio, y por consiguiente, amistoso para el resto de ficheros del directorio.

private: ¡eso no se toca!

La palabra clave **private** significa que nadie puede acceder a ese miembro excepto a través de los métodos de esa clase. Otras clases del mismo paquete no pueden acceder a miembros **privados**, de forma que es como si se estuviera incluso aislando la clase contra uno mismo. Por otro lado, no es improbable que un paquete esté construido por varias personas que colaboran juntas, de forma que **privado** permite cambiar libremente ese miembro sin necesidad de preocuparse de si el cambio influirá a otras clases del mismo paquete.

El acceso "amistoso" al paquete por defecto proporciona un nivel de ocultación bastante elevado; recuerde, un miembro "amistoso" es inaccesible para el usuario del paquete. Esto está bien, dado que el acceso por defecto es el que se usa normalmente (y el que se lograría si se olvida añadir

algún control de acceso). Por consiguiente, uno generalmente pensaría en lo referente al acceso a los miembros de un programa, que habría que hacer éstos explícitamente **públicos**, y como resultado, puede que inicialmente no se piense en usar la palabra clave **private** a menudo. (Lo cual es distinto en C++.) Sin embargo, resulta que el uso consistente de **private** es muy importante, especialmente cuando está involucrada la ejecución multihilo. (Como se verá en el Capítulo 14.)

He aquí un ejemplo del uso de **private**:

```
//: c05:Helado.java
// Demuestra el uso de la palabra clave "private".

class Vainilla {
  private Vainilla() {}
  static Vainilla prepararVainilla() {
    return new Vainilla();
  }
}

public class Helado {
  public static void main(String[] args) {
    //! Vainilla x = new Vainilla();
    Vainilla x = Vainilla.prepararVainilla();
  }
} ///:~
```

Esto muestra un ejemplo de cómo el modificador **privado** resulta útil: se podría querer controlar cómo se crea un objeto y evitar que alguien pueda acceder directamente a un constructor particular (o a todos ellos). En el ejemplo de arriba, no se puede crear un objeto **Vainilla** a través de su constructor; por el contrario, debe invocarse al método **prepararVainilla( )**[2].

Puede declararse **privado** cualquier método del que tengamos la seguridad de que no es más que un método "ayudante" para esa clase, para asegurar que no se use accidentalmente en ningún otro lugar del paquete, y por consiguiente, prohibir a uno mismo cambiar o eliminar el método. Construir un método **privado** garantiza que se conserve esta opción.

Lo mismo es válido para un campo **privado** dentro de una clase. A menos que se deba exponer la implementación subyacente (lo cual es una situación mucho más rara de lo que se podría pensar), deberían hacerse **privados** todos los campos. Sin embargo, sólo porque una referencia a un objeto sea **privado** dentro de su clase, no es imposible que cualquier otro objeto pueda tener una referencia **pública** al mismo objeto. (Apéndice A para aspectos relativos al "uso de alias".)

---

[2] Hay otro efecto en este caso: dado que el constructor por defecto es el único definido, y éste es **privado**, evitará la herencia de esta clase. (Un aspecto que se detallará en el Capítulo 6.)

protected: "un tipo de amistades"

Entender el especificador de acceso **protegido** supone ir algo más allá. En primer lugar, uno debería ser consciente de que no necesita entender esta sección para continuar a lo largo de este libro hasta llegar a la herencia (Capítulo 6). Pero de manera comparativa, he aquí una breve descripción y ejemplo utilizando **protected**.

La palabra clave **protected** está relacionada con un concepto denominado *herencia*, que toma una clase existente y le añade nuevos miembros sin tocar la clase ya existente, a la que se denomina *clase base*. También se puede cambiar el comportamiento de los miembros existentes de la clase. Para heredar de una clase existente, se dice que la nueva clase **hereda** de una ya existente, como:

```
class FOO extends Bar {
```

El resto de la definición de la clase es exactamente igual.

Si se crea un nuevo paquete y se hereda desde una clase de otro paquete, los únicos miembros a los que se tiene acceso son los miembros **públicos** del paquete original. (Por supuesto, si se lleva a cabo la herencia dentro del *mismo* paquete, se tiene el acceso de paquete normal a todos los miembros "amistosos".) Algunas veces, el creador de la clase base desea tomar un miembro particular y garantizar el acceso a las clases derivadas, pero no a todo el mundo. Esto es lo que hace el modo **protegido**. Si se hiciera referencia de nuevo al fichero **Galleta.java**, la siguiente clase *no puede* acceder al miembro "amistoso":

```
//: c05:GalletaChocolate.java
// No puede acceder a un miembro amistoso.
// de otra clase.
import c05.postre.*;

public class GalletaChocolate extends Galleta {
  public GalletaChocolate() {
   System.out.println(
     "Constructor de GalletaChocolate");
  }
  public static void main(String[] args) {
    GalletaChocolate x = new GalletaChocolate();
    //! x.morder(); // No se puede acceder a morder.
  }
} ///:~
```

Una de las cosas más interesantes de la herencia es que si existe un método **morder( )** en la clase **Galleta**, también existe en cualquier clase heredada de **Galleta**. Pero dado que **morder( )** es "amistoso" para los otros paquetes, no podrá utilizarse en éstos. Por supuesto, se puede hacer que sea **público**, pero entonces todo el mundo tendría acceso y quizás eso no es lo que se desea. Si se cambia la clase **Galleta**, como sigue:

```
public class Galleta {
```

```
    public Galleta() {
      System.out.println("Constructor de galletas");
    }
    protected void morder() {
      System.out.println("morder");
}
} ///:~
```

entonces **morder( )** sigue teniendo acceso "amistoso" dentro del paquete **postre**, pero también es accesible a cualquiera que herede de **Galleta**. Sin embargo, *no* es **público**.

Interfaz e implementación

El control de accesos se suele denominar *ocultación de la información*. Al hecho de envolver datos y miembros dentro de las clases, en combinación con el ocultamiento de la información, se le suele denominar *encapsulación*[3]. El resultado es un tipo de datos con sus propias características y comportamientos.

El control de accesos pone límites dentro de un tipo de datos por dos razones importantes. La primera es establecer qué es lo que pueden y lo que no pueden usar los programadores cliente. Se pueden construir los mecanismos internos dentro de la estructura sin tener que preocuparse de que los programadores clientes traten de manipular accidentalmente las interioridades como parte de la interfaz, que es lo que deberían estar usando.

Esto nos presenta directamente en la segunda razón, que es separar la interfaz de la implementación. Si la estructura se utiliza en un conjunto de programas, los programadores clientes no pueden hacer nada más que enviar mensajes al interfaz **público**, entonces es posible cambiar cualquier cosa que *no* sea **público** (por ejemplo, "amistoso", **protegido** o **privado**) sin necesidad de requerir modificaciones en el código cliente.

Ahora nos encontramos en el mundo de la programación orientada a objetos, donde una **clase** describe, de hecho, "una clase de objetos", tal y como se describiría una clase de pescados o una clase de pájaros. Cualquier objeto que pertenezca a esta clase compartirá estas características y comportamientos. La clase es una descripción de lo que parecen y de cómo se comportan los objetos de este tipo.

En el lenguaje original de POO, Simula-67, la palabra clave **class** se utilizaba para describir un nuevo tipo de datos. La misma palabra clave se ha venido utilizando en la mayoría de lenguajes orientados a objetos. Éste es el punto más importante de todo el lenguaje: la creación de nuevos tipos de datos que son más que simples cajas contenedoras de datos y métodos.

La clase es el concepto fundamental en Java. Es una de las palabras clave que *no* se pondrá en negrita en este libro —pues resultaría molesto hacerlo con una palabra que se repite tan a menudo.

[3] Sin embargo, la gente suele denominar "encapsulación" únicamente al ocultamiento de información.

Por claridad, puede que se prefiera un estilo de creación de clases que ponga los miembros **públicos** al principio, seguidos de los miembros **protegidos**, amistosos y **privados**. La ventaja es que el usuario de la clase puede ir leyendo de arriba hacia abajo y ver primero lo que más le importa (los miembros **públicos**, que es a los que puede acceder desde fuera del archivo) y dejar de leer cuando encuentre los miembros no **públicos**, que son parte de la implementación interna.

```
public class x {
  public void pub1() { /* . . . */}
  public void pub2() { /* . . . */}
  public void pub3() { /* . . . */}
  private void priv1() { /* . . . */ }
  private void priv2() { /* . . . */ }
  private void priv3() { /* . . . */ }
  private int i;
  // . . .
}
```

Esto la hará simplemente un poco más fácil de leer, puesto que la interfaz y la implementación siguen estando entremezclados. Es decir, sigue siendo necesario ver el código fuente —la implementación, porque está justo ahí, dentro de la clase. Además, la documentación en forma de comentarios soportada por javadoc (descrito en el Capítulo 2) resta la importancia de la legibilidad del código para el programador cliente. Mostrar la interfaz al consumidor de una clase es verdaderamente el trabajo del *navegador de clases* o *class browser*, una herramienta cuyo trabajo es mirar en todas las clases disponibles y mostrar lo que se puede hacer con ellas (por ejemplo, qué miembros están disponibles) de forma útil. Para cuando se lea el presente texto, cualquier buena herramienta de desarrollo Java debería incluir este tipo de navegadores.

Acceso a clases

En Java, los especificadores de acceso pueden usarse también para determinar qué clases estarán disponibles *dentro* de una biblioteca para los usuarios de esa biblioteca. Si se desea que una clase esté disponible para un programador cliente, se coloca la palabra clave **public** en algún lugar antes de la llave de apertura del cuerpo de la clase. Esto controla si el programador cliente puede incluso crear objetos de esa clase.

Para controlar el acceso a una clase, debe aparecer el especificador antes de la palabra clave **class**. Por consiguiente, se puede decir:

```
public class Componente {
```

Ahora, si el nombre de la biblitoeca es **mibiblioteca**, cualquier programador cliente puede acceder a **Componente** diciendo

```
import mibiblioteca.Componente;
```

o

```
import mibiblioteca.*;
```

Sin embargo, hay un conjunto de restricciones extra:

1. Solamente puede haber una clase **pública** por cada unidad de compilación o fichero. La idea es que cada unidad de compilación tenga una única interfaz pública representada por esa clase **pública**. Puede tener tantas clases "amistosas" de soporte como se desee. Si se quiere tener más de una clase **pública** dentro de una unidad de compilación, el compilador mostrará un mensaje de error.

2. El nombre de la clase **pública** debe coincidir exactamente con el nombre del archivo que contenga la unidad de compilación, incluyendo las mayúsculas. Por tanto, para **componente**, el nombre del archivo debe ser **Componente.java** y no **componente.java** o **COMPONENTE.java**. De nuevo, si éstos tampoco coinciden, se obtendrá también un error de tiempo de compilación.

3. Es posible, aunque no habitual, que exista alguna unidad de compilación sin ninguna clase **pública**. En este caso, se puede dar al archivo el nombre que se desee.

¿Qué ocurre si se tiene una clase dentro de **mibiblioteca** que se está utilizando para llevar a cabo las tareas que hace **Componente** o cualquier otra clase **pública** de **mibiblioteca**? Nadie desea llegar hasta el punto de tener que crear documentación para el programador cliente, y pensar que algún tiempo después podría desearse cambiar completamente las cosas y arrancar todas esas clases, para sustituirlas por otras. Para tener esta flexibilidad, hay que asegurar que ningún programador cliente se vuelva dependiente de unos detalles de implementación particulares incluidos dentro de **mibiblioteca**. Para lograr esto, simplemente se quita la palabra **public** de la clase, en cuyo caso se convierte en "amistosa". (La clase puede usarse únicamente dentro de ese paquete.)

Fíjese que una clase no puede ser **privada** (pues esto la convertiría en inaccesible para alguien que no sea la propia clase), ni **protegida**[4]. Por tanto, sólo se tienen dos opciones para los accesos a clases: "amistosa" o **pública**. Si no se desea que nadie más tenga acceso a esa clase, se pueden hacer todos los constructores **privados**, evitando así que nadie más que uno mismo pueda crear un objeto de esa clase[5], dentro de un miembro **estático** de la clase. He aquí un ejemplo:

```
//: c05:Almuerzo.java
// Muestra el funcionamiento de los especificadores de acceso a clases.
// Hace una clase verdaderamente privada
// con constructores privados:

class Sopa {
  private Sopa() {}
  // (1) Permitir la creación a través de un método estático:
  public static Sopa hacerSopa() {
    return new Sopa();
  }
  // (2) Crear un objeto estático nuevo y
  // devolver una referencia bajo demanda.
```

[4] De hecho, una *clase interna* puede ser **privada** o **protegida**, pero se trata de un caso especial. Éstos se presentarán en el Capítulo 7.

[5] También se puede hacer esto por herencia (Capítulo 6) desde esa clase.

```
    // (El patrón "singular"):
    private static Sopa ps1 = new Sopa();
    public static Sopa acceso() {
        return ps1;
    }
    public void f() {}
}

class Bocadilo { // Usa Almuerzo
    void f() { new Almuerzo(); }
}

// Sólo se permite una clase pública por fichero:
public class Almuerzo {
    void prueba() {
        // ¡Esto no se puede hacer! Constructor privado:
        //! Sopa priv1 = new Sopa();
        Sopa priv2 = Sopa.hacerSopa();
        Bocadillo f1 = new Bocadillo();
        Sopa.acceso().f();
    }
} ///:~
```

Hasta ahora, la mayoría de métodos devolvían **void** o un tipo primitivo, por lo que la definición:

```
public static Sopa acceso() {
  return ps1;
}
```

podría parecer algo confusa a primera vista. La palabra antes del nombre del método (**acceso**) indica qué devuelve el método. Hasta la fecha, ésta ha sido la mayoría de las veces vacía (**void**), que quiere decir que no se devuelve nada. Pero también se puede devolver una referencia a un objeto, que es lo que ocurre aquí. Este método devuelve una referencia a un objeto de la clase **Sopa**.

La **clase Sopa** muestra como evitar la creación directa de una clase haciendo **privados** todos los constructores. Recuérdese que si no se crea al menos un constructor explícitamente, se creará automáticamente el constructor por defecto (un constructor sin parámetros). Si se escribiera el constructor por defecto, éste no se creará automáticamente. Al hacerlo **privado**, nadie puede crear un objeto de esa clase. Pero ahora ¿cómo puede alguien usarla? El ejemplo de arriba presenta dos opciones. En primer lugar se crea un método **estático** que crea un nuevo objeto **Sopa** y devuelve una referencia al mismo. Esto podría ser útil si se desea hacer alguna operación extra con la **Sopa** antes de devolverla, o si se desea mantener la cuenta de cuántos objetos **Sopa** crear (quizás para restringir la población de objetos de este tipo).

La segunda opción usa lo que se denomina un *patrón de diseño*, que se describe en *Thinking in Patterns with Java*, descargable de *www.BruceEckel.com*. Este patrón en particular se denomina un "singular", porque sólo permite la creación de un único objeto. El objeto de clase **Sopa** se crea como

un miembro **estático privado** de **Sopa**, por lo que hay uno y sólo uno, y solamente se puede conseguir a través del método **público** de nombre **acceso( )**.

Como se mencionó previamente, si no se desea poner un modificador de acceso para el acceso a una clase, éste es por defecto "amistoso". Esto significa que cualquier otra clase del paquete puede crear un objeto de esa clase, pero no desde fuera del paquete. (Recuérdese que todos los archivos del mismo directorio que no tengan declaraciones explícitas de **paquete** son implícitamente parte del paquete por defecto de ese directorio.) Sin embargo, si un miembro **estático** de esa clase es **público**, el programador cliente puede seguir accediendo al miembro **estático** incluso aunque no pueda crear un objeto de esa clase.

Resumen

En cualquier relación es importante tener unos límites que sean respetados por todas las partes involucradas. Cuando se crea una biblioteca, se establece una relación con el usuario de esa biblioteca —el programador cliente— que es otro programador, que en vez de esto, se encarga unir diverso código para construir una aplicación, o bien de utilizar su biblioteca para construir una aplicación aún más grande.

Sin reglas, los programadores cliente pueden hacer lo que quieran con todos los miembros de una clase, incluso si se desea que no manipulen directamente algunos de estos miembros. Todo aparece desnudo al mundo.

Este capítulo revisaba cómo se construyen clases a partir de bibliotecas; en primer lugar, se explica cómo se empaquetan clases dentro de una biblioteca, y en segundo, cómo controla la clase el acceso a sus miembros.

Se estima que un proyecto de programación en C se empieza a romper entre las 50K y las 100K líneas porque C tiene un único "espacio de nombres" y los nombres empiezan a colisionar, causando una sobrecarga extra de gestión. En Java, la palabra clave **package**, el esquema de nombrado de paquetes *(package)* y la palabra clave *import* dan un control completo sobre los nombres, de manera que se evita de manera sencilla el aspecto de posibles colisiones entre nombres.

Hay dos razones por las que controlar el acceso a los miembros. El primero es mantener las manos de los usuarios alejadas de lo que no deberían tocar; las herramientas que son necesarias para las maquinaciones internas de los tipos de datos, pero no forman parte de la interfaz que los usuarios necesitan para resolver sus problemas. Por tanto, hacer los métodos y campos **privados** es un servicio a los usuarios porque pueden ver fácilmente qué es importante para ellos y qué pueden ignorar. Esto simplifica su grado de entendimiento de la clase.

La segunda y más importante razón para controlar el acceso es permitir al diseñador de bibliotecas cambiar los funcionamientos internos de la clase sin tener que preocuparse de cómo afectará esto al programador cliente. Uno podría construir una clase inicialmente de una forma, y después descubrir que reestructurando el código se logra un aumento considerable de velocidad. Si la interfaz y la implementación están claramente separados y protegidos, se puede acometer este cambio sin forzar al usuario a reescribir su código.

Los modificadores de accesos dan en Java un control muy valioso al creador de la clase. Los usuarios de la clase pueden ver clara y exactamente qué es lo que pueden usar y qué ignorar. Y lo que es más importante, la capacidad para asegurar que ningún usuario se vuelva dependiente de ninguna parte de la implementación subyacente de una clase. Si se conoce ésta, como creador de la misma, se puede cambiar la implementación subyacente con el conocimiento de que ningún programador cliente se verá afectado por los cambios, pues éstos no pueden acceder a esa parte de la clase.

Cuando se tiene la capacidad de cambiar la implementación subyacente, no sólo se puede mejorar su diseño más tarde, sino que también se tiene la libertad de cometer errores. Sin que importe lo cuidadosamente que se haga la planificación y el diseño, se cometerán errores. Sabiendo que cometer estos errores significan seguro que uno experimentará más, aprenderá mejor y acabará antes su proyecto.

La interfaz pública de una clase es lo que el usuario *de hecho, ve,* de forma que conseguir que es lo más importante de una clase es acabar haciéndola "bien" durante el análisis y el diseño. E incluso eso permite alguna libertad de acción de cara al cambio. Si no se logra una interfaz la primera vez, se pueden *añadir* nuevos métodos, siempre que no se elimine ninguno que los programadores cliente se hayan podido usar en sus códigos.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *www.BruceEckel.com.*

1. Escribir un programa que cree un objeto **ListaArray** sin exportaciones explícitas de **java.util.***.

2. En la sección "El paquete: la unidad de biblioteca", cambiar los códigos de fragmento relacionados con **mipaquete** en un conjunto de ficheros Java compilables y ejecutables.

3. En la sección "Colisiones", cambiar los fragmentos de código por un programa, y verificar que verdaderamente se dan colisiones.

4. Generalizar la clase **P** definida en este capítulo añadiendo todas las versiones sobrecargadas de **rint( )** y **rintln( )** necesarias para manejar todos los tipos básicos de Java.

5. Cambiar la sentencia import de **PruebaAfirmacion.java** para habilitar y deshabilitar el mecanismo de afirmaciones.

6. Crear una clase con miembros de datos y métodos **públicos**, **privados**, **protegidos** y "amistosos". Crear un objeto de esa clase y ver qué tipo de mensajes de compilación se obtienen al intentar acceder a todos los miembros de la clase. Ser conscientes de que las clases del mismo directorio son parte del paquete "por defecto".

7. Crear una clase con datos **protegidos**. Crear una segunda clase en el mismo archivo con un método que manipule los datos **protegidos** de la primera clase.

8. Cambiar la clase **Galleta** como se especifica en la sección "**protected**: «tipo de amistad»". Verificar que **morder( )** no es **público**.

9. En la sección titulada "Acceso a clases" se encontrarán fragmentos de código que describen **mibilioteca** y **Componente**. Crear esta biblioteca y posteriormente crear un **Componente** en una clase que no sea parte del paquete **mibiblioteca**.

10. Crear un nuevo directorio y editar la variable CLASSPATH para que incluya ese nuevo directorio. Copiar el archivo **P.class** (producido al compilar **com.burceeckel.herramientas.P.java**) al nuevo directorio y cambiar los nombres del fichero, la clase **P** y los nombres de los métodos. (A lo mejor también se desea añadir alguna salida adicional para ver cómo funciona.) Crear otro programa en un directorio diferente que haga uso de la nueva clase.

11. Siguiendo la forma del ejemplo **Almuerzo.java**, crear una clase denominada **GestorConexion** que gestione un array fijo de objetos **Conexión**. El programador cliente no debe ser capaz de crear explícitamente objetos **Conexión**, sino que solamente puede crear objetos a través de un método estático de **GestorConexion**. Cuando el **GestorConexion** se quede sin objetos, devolverá una referencia **null**. Probar las clases de **main( )**.

12. Crear el siguiente fichero en el directorio c05/local (presumiblemente en el CLASSPATH):

```
///: c05:local:ClaseEmpaquetada.java
package c05.local;
class ClaseEmpaquetada {
  public ClaseEmpaquetada() {
    System.out.println (
      "Creando una clase empaquetada");
  }
}///:~
```

Posteriormente, crear el directorio siguiente en un directorio distinto:

```
///: c05:exterior:Exterior.java
package c05.exterior;
import c05.local.*;
public class Exterior {
  public static void main (String[] args) {
      ClaseEmpaquetada ce = new ClaseEmpaquetada();
  }
} ///:~
```

Explicar por qué el compilador genera un error. ¿Hacer que la clase **Exterior** sea parte del paquete **c05.local** cambiaría algo?

6: Reutilizando clases

Una de las características más atractivas de Java es la reutilización de código. Pero para ser revolucionario, es necesario poder hacer muchísimo más que copiar código y cambiarlo.

Este es el enfoque que se utiliza en los lenguajes procedurales como C, pero no ha funcionado muy bien. Como todo en Java, la solución está relacionada con la clase. Se reutiliza código creando nuevas clases, pero en vez de crearlas de la nada, se utilizan clases ya existentes que otra persona ya ha construido y depurado.

El truco es usar clases sin manchar el código existente. En este capítulo se verán dos formas de lograrlo. La primera es bastante directa: simplemente se crean objetos de las clase existente dentro de la nueva clase. A esto se le llama *composición*, porque la clase nueva está compuesta de objetos de clases existentes. Simplemente se está reutilizando la funcionalidad del código, no su forma.

El segundo enfoque es más sutil. Crea una nueva clase como *un tipo de* una clase ya existente. Literalmente se toma la forma de la clase existente y se le añade código sin modificar a la clase ya existente. Este acto mágico se denomina *herencia*, y el compilador hace la mayoría del trabajo. La herencia es una de las clases angulares de la programación orientada a objetos y tiene implicaciones adicionales como se verá en el Capítulo 7.

Resulta que mucha de la sintaxis y comportamiento son similares, tanto para la herencia, como para la composición (lo cual tiene sentido porque ambas son formas de construir nuevos tipos a partir de tipos existentes). En este capítulo, se aprenderá sobre estos mecanismos de reutilización de código.

Sintaxis de la composición

Hasta ahora, la composición se usaba con bastante frecuencia. Simplemente se ubican referencias a objetos dentro de nuevas clases. Por ejemplo, suponga que se desea tener un objeto que albergue varios objetos de tipo **cadena de caracteres**, un par de datos primitivos, y un objeto de otra clase. En el caso de los objetos no primitivos, se ponen referencias dentro de la nueva clase, pero se definen los datos primitivos directamente:

```
//: c06:Aspersor.java
// Composición para la reutiliación de código.

class FuenteAgua {
  private String s;
  FuenteAgua() {
    System.out.println("FuenteAgua()");
    s = new String("Construida");
  }
  public String toString() { return s; }
```

```
}

public class Aspersor {
  private String valvula1, valvula2, valvula3, valvula4;
  FuenteAgua fuente;
  int i;
  float f;
  void escribir() {
    System.out.println("valvula1 = " + valvula1);
    System.out.println("valvula2 = " + valvula2);
    System.out.println("valvula3 = " + valvula3);
    System.out.println("valvula4 = " + valvula4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("fuente = " + fuente);
  }
  public static void main(String[] args) {
    Aspersor x = new Aspersor();
    x.escribir();
  }
} ///:~
```

Uno de los métodos definidos en **FuenteAgua( )** es especial: **toString( )**. Se aprenderá más adelante que todo objeto no primitivo tiene un método **toString( )**, y es invocado en situaciones especiales cuando el compilador desea obtener un objeto como cadena de caracteres. Por tanto, en la expresión:

```
System.out.println("fuente = " + fuente);
```

el compilador ve que se está intentando añadir un objeto **String** ("**fuente =**") a un objeto **FuenteAgua.** Esto no tiene sentido porque sólo se puede "añadir" un **String** a otro **String**, por lo que dice: "¡Convertiré **fuente** en un **String** invocando a **toString( )**! Después de hacer esto se pueden combinar los dos objetos de tipo **String** y pasar el **String** resultante a **Sytem.out.println( )**. Siempre que se desee este comportamiento con una clase creada, sólo habrá que escribir un método **toString( )**.

A primera vista, uno podría asumir —siendo Java tan seguro y cuidadoso como es— que el compilador podría construir automáticamente objetos para cada una de las referencias en el código de arriba; por ejemplo, invocando al constructor por defecto para **FuenteAgua** para inicializar **fuente**. La salida de la sentencia de impresión es de hecho:

```
valvula1 = null
valvula2 = null
valvula3 = null
valvula4 = null
i = 0
f = 0.0
fuente = null
```

Los datos primitivos son campos de una clase que se inicializan automáticamente a cero, como se indicó en el Capítulo 2. Pero las referencias a objetos se inicializan a **null**, y si se intenta invocar a métodos de cualquiera de ellos, se sigue obteniendo una excepción. De hecho, es bastante bueno (y útil) poder seguir imprimiéndolos sin lanzar excepciones.

Tiene sentido que el compilador no sólo cree un objeto por defecto para cada referencia, porque eso conllevaría una sobrecarga innecesaria en la mayoría de los casos. Si se quieren referencias inicializadas, se puede hacer:

1. En el punto en que se definen los objetos. Esto significa que siempre serán inicializados antes de invocar al constructor.
2. En el constructor de esa clase.
3. Justo antes de que, de hecho, se necesite el objeto. A esto se le llama *inicialización perezosa*. Puede reducir la sobrecarga en situaciones en las que no es necesario crear siempre el objeto.

A continuación, se muestran los tres enfoques:

```
//: c06:Banio.java
// Inicialización de constructores con composición.

class Jabon {
  private String s;
  Jabon() {
    System.out.println("Jabon()");
    s = new String("Construido");
  }
  public String toString() { return s; }
}

public class Banio {
  private String
    // Inicializando en el momento de la definición:
    s1 = new String("Contento"),
    s2 = "Contento",
    s3, s4;
  Jabon pastilla;
  int i;
  float juguete;
  Banio() {
    System.out.println("Dentro del banio()");
    s3 = new String("Gozo");
    i = 47;
    juguete = 3.14f;
    pastilla = new Jabon();
  }
```

```
  void escribir() {
    // Inicialización tardía:
    if(s4 == null)
      s4 = new String("Temporal");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("juguete = " + juguete);
    System.out.println("pastilla = " + pastilla);
  }
  public static void main(String[] args) {
    Banio b = new Banio();
    b.escribir();
  }
} ///:~
```

Fíjese que en el constructor **Banio** se ejecuta una sentencia antes de que tenga lugar ninguna inicialización. Cuando no se inicializa en el momento de la definición, sigue sin haber garantías de que se lleve a cabo ningún tipo de inicialización antes de que se envíe un mensaje a una referencia a un objeto —excepto la inevitable excepción en tiempo de ejecución.

He aquí la salida del programa:

```
Dentro del Banio()
Jabon()
S1 = Contento
S2 = Contento
S3 = Gozo
S4 = Gozo
I = 47
Juguete = 3.14
pastilla = Construido
```

Cuando se invoca al método **escribir( )** éste rellena **s4** para que todos los campos estén inicializados correctamente cuando se usen.

Sintaxis de la herencia

La herencia es una parte integral de Java (y de todos los lenguajes de POO en general). Resulta que siempre se está haciendo herencia cuando se crea una clase, pero a menos que se herede explícitamente de otra clase, se hereda implícitamente de la clase raíz estándar de Java **Object**.

La sintaxis para la composición es obvia, pero para llevar a cabo herencia se realiza de distinta forma. Cuando se hereda, se dice: "Esta clase nueva es como esa clase vieja". Se dice esto en el códi-

go dando el nombre de la clase, como siempre, pero antes de abrir el paréntesis del cuerpo de la clase, se pone la palabra clave **extends** seguida del nombre de la *clase base*. Cuando se hace esto, automáticamente se tienen todos los datos miembro y métodos de la clase base. He aquí un ejemplo:

```
//: c06:Detergente.java
// Sintaxis y propiedades de la herencia.

class ProductoLimpieza {
  private String s = new String("Producto de Limpieza");
  public void aniadir(String a) { s += a; }
  public void diluir() { aniadir(" diluir()"); }
  public void aplicar() { aniadir(" aplicar()"); }
  public void frotar() { aniadir(" fregar()"); }
  public void escribir() { System.out.println(s); }
  public static void main(String[] args) {
    ProductoLimpieza x = new ProductoLimpieza();
    x.diluir(); x.aplicar(); x.frotar();
    x.escribir();
  }
}

public class Detergente extends ProductoLimpieza {
  // Cambiar un método:
  public void frotar() {
    aniadir(" Detergente.frotar()");
    super.frotar(); // Llamar a la versión de la clase base
  }
  // Añadir métodos al interfaz:
  public void aclarar() { aniadir(" aclarar()"); }
  // Probar la nueva clase:
  public static void main(String[] args) {
    Detergente x = new Detergente();
    x.diluir();
    x.aplicar();
    x.frotar();
    x.aclarar();
    x.escribir();
    System.out.println("Probando la clase base:");
    ProductoLimpieza.main(args);
  }
} ///:~
```

Esto demuestra un gran número de aspectos. En primer lugar, en el método **aniadir( ) de la clase ProductoLimpieza**, se concatenan **Cadenas de caracteres** a **s** utilizando el operador +=, que

es uno de los operadores (junto con '+') que los diseñadores de Java "sobrecargaron" para que funcionara con **Cadenas de caracteres**.

Segundo, tanto **ProductoLimpieza** como **Detergente** contienen un método **main( )**. Se puede crear un método **main( )** por cada clase que uno cree, y se recomienda codificar de esta forma, de manera que todo el código de prueba esté dentro de la clase. Incluso si se tienen muchas clases en un programa, sólo se invocará al método **main( )** de la clase invocada en la línea de comandos. (Dado que **main( )** es **público**, no importa si la clase a la que pertenece es o no **pública**.) Por tanto, en este caso, cuando se escriba **java Detergente**, se invocará a **Detergente.main( )**. Pero también se puede hacer que **ProductoLimpieza** invoque a **ProductoLimpieza.main( )**, incluso aunque **ProductoLimpieza** no sea una clase **pública**. Esta técnica de poner un método **main( )** en cada clase permite llevar a cabo pruebas para cada clase de manera sencilla. Y no es necesario eliminar el método **main( )** cuando se han acabado las pruebas; se puede dejar ahí por si hubiera que usarlas para otras pruebas más adelante.

Aquí, se puede ver que **Detergente.main( )** llama a **ProductoLimpieza.main( )** explícitamente, pasándole los mismos argumentos de la línea de comandos (sin embargo, se podría pasar cualquier array de **Cadenas de caracteres**).

Es importante que todos los métodos de **ProductoLimpieza** sean **públicos**. Recuerde que si se deja sin poner cualquier modificador de miembro, el miembro será por defecto "amistoso", lo cual permite acceder sólo a los miembros del paquete. Por consiguiente, *dentro de este paquete*, cualquiera podría usar esos métodos si no hubiera modificador de acceso. **Detergente** no tendría problemas, por ejemplo. Sin embargo, si se fuera a heredar desde **ProductoLimpieza** una clase de cualquier otro paquete, ésta sólo podría acceder a las clases **públicas**. Por tanto, al planificar la herencia, como regla general, deben hacerse todos los campos **privados** y todos los miembros **públicos**. (Los miembros **protegidos** también permiten accesos por parte de clases derivadas; esto se aprenderá más adelante.) Por supuesto, en los casos particulares hay que hacer ajustes, pero ésta es una regla útil.

Fíjese que **ProductoLimpieza** tiene un conjunto de métodos en su interfaz: **aniadir( )**, **diluir( )**, **aplicar( )**, **frotar( )** y **escribir( )**. Dado que **Detergente** se *hereda de* **ProductoLimpieza** (mediante la palabra clave **extends**) automáticamente se hace con estos métodos en su interfaz, incluso aunque no se encuentren explícitamente definidos en **Detergente**. Se puede pensar que la herencia, por tanto, es una *reutilización del interfaz.* (La implementación también se hereda, pero esto no es lo importante.)

Como se ha visto en **frotar( )**, es posible tomar un método que se haya definido en la clase base y modificarlo. En este caso, se podría desear llamar al método desde la clase base dentro de la nueva versión. Pero dentro de **frotar( )** no se puede simplemente invocar a **frotar( )**, dado que eso produciría una llamada recursiva, que no es lo que se desea. Para solucionar este problema Java tiene la palabra clave **super** que hace referencia a la "superclase" de la cual ha heredado la clase actual. Por consiguiente, la expresión **super.frotar( )** llama a la versión que tiene la clase base del método **frotar( )**.

Al heredar, uno no se limita a usar los métodos de la clase base. También se pueden añadir nuevos métodos a la clase derivada, exactamente de la misma manera que se introduce un método en una clase: simplemente se definen. El método **aclarar( )** es un ejemplo de esta afirmación.

En **Detergente.main( )** se puede ver que, para un objeto **Detergente,** se puede invocar a todos los métodos disponibles, también en **ProductoLimpieza** y en **Detergente** (por ejemplo, **aclarar( )**).

Inicializando la clase base

Dado que ahora hay dos clases involucradas —la clase base y la clase derivada— en vez de simplemente una, puede ser un poco confuso intentar imaginar el objeto resultante producido por una clase derivada. Desde fuera, parece que la nueva clase tiene la misma interfaz que la clase base, y quizás algunos métodos y campos adicionales. Pero la herencia no es una simple copia de la interfaz de la clase base. Cuando se crea un objeto de la clase derivada, éste contiene dentro de él un *sub objeto* de la clase base. Este subobjeto es el mismo que si se hubiera creado un objeto de la clase base en sí. Es simplemente que, desde fuera, el subobjeto de la clase base está envuelto dentro del objeto de la clase derivada.

Por supuesto, es esencial que el subobjeto de la clase base se inicialice correctamente y sólo hay una forma de garantizarlo: llevar a cabo la inicialización en el constructor, invocando al constructor de la clase base, que tiene todo el conocimiento y privilegios apropiados para llevar a cabo la inicialización de la clase base. Java inserta automáticamente llamadas al constructor de la clase base en el constructor de la clase derivada. El ejemplo siguiente muestra este funcionamiento con tres niveles de herencia:

```
//: c06:Animacion.java
// Llamadas al constructor durante la herencia.

class Arte {
  Arte() {
    System.out.println("Constructor de arte");
  }
}

class Dibujo extends Arte {
  Dibujo() {
    System.out.println("Constructor de dibujo");
  }
}

public class Animacion extends Dibujo {
  Animacion() {
    System.out.println("Constructor de animacion");
  }
  public static void main(String[] args) {
    Animacion x = new Animacion();
  }
} ///:~
```

La salida de este programa muestra las llamadas automáticas:

```
Constructor de arte
Constructor de dibujo
Constructor de animacion
```

Se puede ver que la construcción se da desde la base "hacia fuera", de forma que se inicializa la clase base antes de que los constructores de la clase derivada puedan acceder a ella.

Incluso si no se crea un constructor para **Animacion( )**, el compilador creará un constructor por defecto que invoque al constructor de la clase base.

Constructores con parámetros

El ejemplo de arriba tiene constructores por defecto; es decir, no tienen ningún parámetro. Para el compilador es fácil invocarlos porque no hay ningún problema que resolver respecto al paso de parámetros. Si una clase no tiene parámetros por defecto, o si se desea invocar a un constructor de una clase base que tiene parámetros, hay que escribir explícitamente la llamada al constructor de la clase base usando la palabra clave **super** y la lista de parámetros apropiada:

```
//: c06:Ajedrez.java
// Herencia, constructores y parámetros.

class Juego {
  Juego(int i) {
    System.out.println("Constructor de juego");
  }
}

class JuegoMesa extends Juego {
  JuegoMesa(int i) {
    super(i);
    System.out.println("Constructor de JuegoMesa");
  }
}

public class Ajedrez extends JuegoMesa {
  Ajedrez() {
    super(11);
    System.out.println("Constructor de Ajedrez");
  }
  public static void main(String[] args) {
    Ajedrez x = new Ajedrez();
  }
} ///:~
```

Si no se invoca al constructor de la clase base de **JuegoMesa( )**, el compilador se quejará al no poder encontrar un constructor de la forma **Juego( )**. Además, la llamada al constructor de la clase base *debe* ser lo primero que se haga en el constructor de la clase derivada. (El compilador así lo recordará cuando no se haga correctamente.)

Capturando excepciones del constructor base

Como se acaba de indicar, el compilador obliga a ubicar la llamada al constructor de la clase base, primero dentro del cuerpo del constructor de la clase derivada. Esto simplemente quiere decir que no puede aparecer nada antes de esta llamada. Como se verá en el Capítulo 10, esto también evita que un constructor de una clase derivada capture excepciones que provengan de una clase base. Esto puede suponer un inconveniente en algunas ocasiones.

Combinando la composición y la herencia

Es muy frecuente usar la composición y la herencia juntas. El ejemplo siguiente muestra la creación de una clase más compleja, utilizando tanto la herencia como la composición, junto con la inicialización necesaria del constructor:

```
//: c06:PonerMesa.java
// Combinando la composición y la herencia.

class Plato {
  Plato(int i) {
    System.out.println("Constructor de plato");
  }
}

class PlatoCena extends Plato {
  PlatoCena(int i) {
    super(i);
    System.out.println(
      "Constructor de PlatoCena");
  }
}

class Utensilio {
  Utensilio(int i) {
    System.out.println("Constructor de utensilio");
  }
}

class Cuchara extends Utensilio {
```

```
  Cuchara(int i) {
    super(i);
    System.out.println("Constructor de cuchara");
  }
}

class Tenedor extends Utensilio {
  Tenedor(int i) {
    super(i);
    System.out.println("Constructor de tenedor");
  }
}

class Cuchillo extends Utensilio {
  Cuchilllo(int i) {
    super(i);
    System.out.println("Constructor de cuchillo");
  }
}

// Una manera costrumbrista de hacer algo:
class Costumbre {
  Costumbre(int i) {
    System.out.println("Constructor de costumbre");
  }
}

public class PonerMesa extends Costumbre {
  Cuchara cc;
  Tenedor tnd;
  Cuchillo cch;
  PlatoCena pc;
  PonerMesa(int i) {
    super(i + 1);
    cc = new Cuchara(i + 2);
    tnd = new Tenedor(i + 3);
    cch = new Cuchillo(i + 4);
    pc = new PlatoCena(i + 5);
    System.out.println(
      "Constructor de PonerMesa");
  }
  public static void main(String[] args) {
    PonerMesa x = new PonerMesa(9);
  }
} ///:~
```

Cuando el compilador obliga a inicializar la clase base, y requiere que se haga justo al principio del constructor, no se asegura de que inicialicemos los objetos miembro, por lo que es conveniente prestar especial atención a esto.

Garantizar una buena limpieza

Java no tiene el concepto de método *destructor* de C++. Este método se invoca automáticamente al destruir un objeto. La razón de su ausencia es probablemente que en Java lo habitual es simplemente olvidarse de esos objetos, más que destruirlos, permitiendo que el recolector de basura reclame esta memoria cuando sea necesario.

En muchas ocasiones, esto es bueno, pero hay veces en las que una clase tiene que hacer algunas actividades durante su vida que requieren de limpieza. Como se mencionó en el Capítulo 4, no se puede saber cuándo se invocará al recolector de basura, o incluso, si éste será invocado. Por tanto, si se desea que se limpie algún espacio para una clase, hay que escribir explícitamente un método especial que lo haga, y asegurarse de que el programador cliente sepa que hay que invocar a este método. Por encima de esto —como se describe en el Capítulo 10 ("Manejo de Errores con Excepciones")— hay que protegerse de las excepciones poniendo este tipo de limpieza en una cláusula **finally**.

Considere un ejemplo de un sistema de diseño asistido por computador que dibuja en la pantalla:

```
//: c06:SistemaDAC.java
// Asegurando una limpieza adecuada.
import java.util.*;

class Forma {
  Forma(int i) {
    System.out.println("Constructor de forma");
  }
  void limpiar() {
    System.out.println("Limpieza de forma");
  }
}

class Circulo extends Forma {
  Circulo(int i) {
    super(i);
    System.out.println("Dibujando un circulo");
  }
  void limpiar() {
    System.out.println("Borrando un circulo");
    super.limpiar();
  }
}
```

```
class Triangulo extends Forma {
  Triangulo(int i) {
    super(i);
    System.out.println("Dibujando un triangulo");
  }
  void limpiar() {
    System.out.println("Borrando un triangulo");
    super.limpiar();
  }
}

class Linea extends Forma {
  private int inicio, fin;
  Linea(int inicio, int fin) {
    super(inicio);
    this.inicio = inicio;
    this.fin = fin;
    System.out.println("Dibujando una linea: " +
           inicio + ", " + fin);
  }
  void limpiar() {
    System.out.println("Borrando una linea: " +
           inicio + ", " + fin);
    super.limpiar();
  }
}

public class SistemaDAC extends Forma {
  private Circulo c;
  private Triangulo t;
  private Linea[] lineas = new Linea[10];
  SistemaDAC(int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
      lineas[j] = new Linea(j, j*j);
    c = new Circulo(1);
    t = new Triangulo(1);
    System.out.println("Constructor combinado");
  }
  void limpiar() {
    System.out.println("SistemaDAC.limpiar()");
    // El orden de eliminación es inverso al
    // orden de inicialización
    t.limpiar();
    c.limpiar();
```

```
    for(int i = lineas.length - 1; i >= 0; i--)
      lineas[i].limpiar();
    super.limpiar();
  }
  public static void main(String[] args) {
    SistemaDAC x = new SistemaDAC(47);
    try {
      // Código y manejo de excepciones...
    } finally {
      x.limpiar();
    }
  }
} ///:~
```

Todo en este sistema es algún tipo de **Forma** (que en sí es un tipo de **Objeto** dado que está implícitamente heredada de la clase raíz). Cada clase redefine el método **limpiar( )** de **Forma** además de invocar a la versión de ese método de la clase base haciendo uso de **super**. Las clases **Forma** específicas —**Círculo**, **Triángulo** y **Línea**— tienen todas constructores que "dibujan", aunque cualquier método invocado durante la vida del objeto podría ser el responsable de hacer algo que requiera de limpieza. Cada clase tiene su propio método **limpiar( )** para restaurar cosas a la forma en que estaban antes de que existiera el objeto.

En el método **main( )** se pueden ver dos palabras clave nuevas, y que no se presentarán oficialmente hasta el capítulo 10: **try** y **finally**. La palabra clave **try** indica que el bloque que sigue (delimitado por llaves) es una *región vigilada*, lo que quiere decir que se le da un tratamiento especial. Uno de estos tratamientos especiales consiste en que el código de la cláusula **finally** que sigue a esta región vigilada se ejecuta *siempre*, sin que importe cómo se salga del bloque **try**. (Con el manejo de excepciones, es posible dejar un bloque **try** de distintas formas no ordinarias.) Aquí, la cláusula **finally** dice: "Llama siempre a **limpiar( )** para **x**, sin que importe lo que ocurra". Estas palabras claves se explicarán con detalle en el Capítulo 10.

Fíjese que en el método de limpieza hay que prestar atención también al orden de llamada de los métodos de limpieza de la clase base y los objetos miembros, en caso de que un subobjeto dependa de otro. En general, se debería seguir la forma ya impuesta por el compilador de C++ para sus destructores: en primer lugar se lleva a cabo todo el trabajo de limpieza específico a nuestra clase, en orden inverso de creación. (En general, esto requiere que los elementos de la clase base sigan siendo accesibles.) Después, se llama al método de limpieza de la clase base, como se ha demostrado aquí.

Puede haber muchos casos en los que el aspecto de la limpieza no sea un problema; simplemente se deja actuar al recolector de basura. Pero cuando es necesario hacerlo explícitamente se necesita tanto diligencia como atención.

Orden de recolección de basura

No hay mucho en lo que se pueda confiar en lo referente a la recolección de basura. Puede que ni siquiera se invoque nunca al recolector de basura. Cuando se le invoca, puede reclamar objetos en el orden que quiera. Es mejor no confiar en la recolección de basura para nada que no sea reclamar

memoria. Si se desea que se dé una limpieza, es mejor que cada uno construya sus propios métodos de limpieza, y no confiar en el método **finalize( )**. (Como se mencionó en el Capítulo 4, puede obligarse a Java a invocar a todos los métodos **finalize( )**.)

Ocultación de nombres

Sólo los programadores de C++ podrían sorprenderse de la ocultación de nombres, puesto que funciona distinto en ese lenguaje. Si una clase base de Java tiene un nombre de método sobrecargado varias veces, la redefinición de ese nombre de método en la clase derivada *no* esconderá ninguna de las versiones de la clase base. Por consiguiente, la sobrecarga funciona independientemente de si el método se definió en el nivel actual o en una clase base:

```
//: c06:Ocultar.java
// Sobrecargando un nombre de método de una clase base
// en una clase derivada que no oculta
// las versiones de la clase base.

class Homer {
  char realizar(char c) {
    System.out.println("realizar(char)");
    return 'd';
  }
  float realizar(float f) {
    System.out.println("realizar(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void realizar(Milhouse m) {}
}

class Ocultar {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.realizar(1); // realizar(float) usado
    b.realizar('x');
    b.realizar(1.0f);
    b.realizar(new Milhouse());
  }
} ///:~
```

Como se verá en el siguiente capítulo, es bastante más común reescribir métodos del mismo nombre utilizando exactamente el mismo nombre, parámetros y tipo de retorno que en la clase base. De otra manera pudiera ser confuso (que es la razón por la que C++ no permite esto, para evitar que se haga lo que probablemente es un error).

Elección entre composición y herencia

Tanto la composición como la herencia, permiten ubicar subobjetos dentro de una nueva clase. Habría que preguntarse por la diferencia entre ambas, y cuándo elegir una en vez de la otra.

La composición suele usarse cuando se quieren mantener las características de una clase ya existente dentro de la nueva, pero no su interfaz. Es decir, se empotra un objeto de forma que se puede usar para implementar su funcionalidad en la nueva clase, pero el usuario de la nueva la clase ve la interfaz que se ha definido para la nueva clase en vez de la interfaz del objeto empotrado. Para lograr este efecto, se empotran objetos **privados** de clases existentes dentro de la nueva clase.

En ocasiones, tiene sentido permitir al usuario de la clase acceder directamente a la composición de la nueva clase; es decir, hacer a los objetos miembro **públicos**. Los objetos miembro usan por sí mismos la ocultación de información, de forma que esto es seguro. Cuando el usuario sabe que se está ensamblando un conjunto de partes, construye una interfaz más fácil de entender. Un objeto **coche** es un buen ejemplo:

```
//: c06:Coche.java
// Composición con objetos públicos.

class Motor {
  public void arrancar() {}
  public void acelerar() {}
  public void parar() {}
}

class Rueda {
  public void inflar(int psi) {}
}

class Ventana {
  public void subir() {}
  public void bajar() {}
}

class Puerta {
  public Ventana ventana = new Ventana();
  public void abrir() {}
  public void cerrar() {}
}
```

```
public class Coche {
  public Motor motor = new Motor();
  public Rueda[] rueda = new Rueda[4];
  public Puerta izquierda = new Puerta(),
       derecha = new Puerta(); // 2-puerta
  public Coche() {
    for(int i = 0; i < 4; i++)
      rueda[i] = new Rueda();
  }
  public static void main(String[] args) {
    Coche coche = new Coche();
    coche.izquierda.ventana.subir();
    coche.rueda[0].inflar(72);
  }
} ///:~
```

Dado que la composición de un coche es parte del análisis del problema (y no simplemente parte del diseño subyacente), hacer sus miembros **públicos** ayuda al entendimiento por parte del programador cliente de cómo usar la clase, y requiere menos complejidad de código para el creador de la clase. Sin embargo, hay que ser consciente de que éste es un caso especial y en general los campos se harán **privados**.

Cuando se hereda, se toma una clase existente y se hace una versión especial de la misma. En general, esto significa que se está tomando una clase de propósito general y especializándola para una necesidad especial. Simplemente pensando un poco se verá que no tendría sentido componer un coche utilizando un objeto vehículo —un coche no contiene un vehículo, *es* un vehículo. La relación *es-un* se expresa con herencia, y la relación *tiene-un* se expresa con composición.

Protegido (protected)

Ahora que se ha presentado el concepto de herencia, tiene sentido finalmente la palabra clave **protected**. En un mundo ideal, los miembros **privados** siempre serían irrevocablemente **privados**, pero en los proyectos reales hay ocasiones en las que se desea hacer que algo quede oculto del mundo en general, y sin embargo, permitir acceso a miembros de clases derivadas. La palabra clave **protected** es un nodo de pragmatismo. Dice: "Esto es **privado** en lo que se refiere al usuario de la clase, pero está disponible para cualquiera que herede de esta clase o a cualquier otro de este **paquete**. Es decir, **protegido** es automáticamente "amistoso" en Java.

La mejor conducta a seguir es dejar los miembros de datos **privados** —uno siempre debería preservar su derecho a cambiar la implementación subyacente. Posteriormente se puede permitir acceso controlado a los descendientes de la clase a través de los métodos **protegidos**:

```
//: c06:Malvado.java
// La palabra clave protected.
import java.util.*;
```

```
class Villano {
  private int i;
  protected int leer() { return i; }
  protected void poner(int ii) { i = ii; }
  public Villano(int ii) { i = ii; }
  public int valor(int m) { return m*i; }
}

public class Malvado extends Villano {
  private int j;
  public Malvado(int jj) { super(jj); j = jj; }
  public void cambiar(int x) { poner(x); }
} ///:~
```

Se puede ver que **cambiar( )** tiene acceso a **poner( )** porque es **protegido**.

Desarrollo incremental

Una de las ventajas de la herencia es que soporta el *desarrollo incremental* permitiendo introducir nuevo código sin introducir *errores* en el código ya existente. Esto también aísla nuevos fallos dentro del nuevo código. Pero al heredar de una clase funcional ya existente y al añadirle nuevos atributos y métodos (y redefiniendo métodos ya existentes), se deja el código existente —que alguien más podría estar utilizando— intacto y libre de errores. Si se da un fallo, se sabe que éste se encuentra en el nuevo código, que es mucho más corto y sencillo de leer que si hubiera que modificar el cuerpo del código existente.

Es bastante sorprendente la independencia de las clases. Ni siquiera se necesita el código fuente de los métodos para reutilizar el código. Como máximo, simplemente habría que importar el paquete. (Esto es cierto, tanto en el caso de la herencia, como en el de la composición.)

Es importante darse cuenta de que el desarrollo de un programa es un proceso incremental, al igual que el aprendizaje humano. Se puede hacer tanto análisis como se quiera, pero se siguen sin conocer todas las respuestas cuando comienza un proyecto. Se tendrá mucho más éxito —y una realimentación mucho más inmediata— si empieza a "crecer" el proyecto como una criatura evolucionaria, orgánica, en vez de construirlo de un tirón como si fuera un rascacielos de cristal.

Aunque la herencia puede ser una técnica útil de cara a la experimentación, en algún momento, una vez que las cosas se estabilizan es necesario echar un nuevo vistazo a la jerarquía de clases definida intentando encajarla en una estructura con sentido. Recuérdese que bajo todo ello, la herencia simplemente pretende expresar una relación que dice: "Esta nueva clase es un *tipo* de esa otra clase". Al programa no deberían importarle los bits, sino el crear y manipular objetos de varios tipos para expresar un modelo en términos que provengan del espacio del problema.

Conversión hacia arriba

El aspecto más importante de la herencia no es que proporcione métodos para la nueva clase. Es la relación expresada entre la nueva clase y la clase base. Esta relación puede resumirse diciendo: "La nueva clase es *un tipo de* la clase existente".

Esta descripción no es simplemente una forma elegante de explicar la herencia —está soportada directamente por el lenguaje. Como ejemplo, considérese una clase base denominada **Instrumento** que representa los instrumentos musicales, y una clase derivada denominada **Viento**. Dado que la herencia significa que todos los métodos de la clase base también están disponibles para la clase derivada, cualquier mensaje que se pueda enviar a la clase base podrá ser también enviado a la clase derivada. Si la clase **Instrumento** tiene un método **tocar( )**, también lo tendrán los instrumentos **Viento**. Esto significa que se puede decir con precisión que un objeto **Viento** es también un tipo de **Instrumento**. El ejemplo siguiente muestra cómo soporta este concepto el compilador:

```
//: c06:Viento.java
// Herencia y conversión hacia arriba.
import java.util.*;

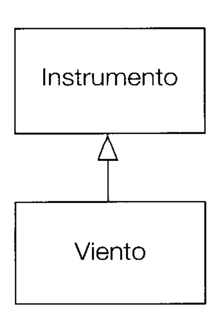
class Instrumento {
  public void tocar() {}
  static void afinar(Instrumento i) {
    // ...
    i.tocar();
  }
}

// Los objetos de viento son instrumentos
// porque tienen la misma interfaz:
class Viento extends Instrumento {
  public static void main(String[] args) {
    Viento flauta = new Viento();
    Instrumento.afinar(flauta); // Conversión hacia arriba
  }
} ///:~
```

Lo interesante de este ejemplo es el método **afinar( )**, que acepta una referencia a **Instrumento**. Sin embargo, en **Viento.main( )**, se llama al método **afinar( )** proporcionándole una referencia a **Viento**. Dado que Java tiene sus particularidades en la comprobación de tipos, parece extraño que un método que acepte un tipo llegue a aceptar otro tipo, hasta que uno se da cuenta de que un objeto **Viento** es también un objeto **Instrumento**, y no hay método al que pueda invocar **afinar( )** para un **Instrumento** que no esté en **Viento**. Dentro de **afinar( )**, el código funciona para **Instrumento** y cualquier cosa que se derive de **Instrumento**, y al acto de convertir una referencia a **Viento** en una referencia a **Instrumento** se le denomina *hacer conversión hacia arriba*.

¿Por qué "conversión hacia arriba"?

La razón para el término es histórica, y se basa en la manera en que se han venido dibujando tradicionalmente los diagramas de herencia: con la raíz en la parte superior de la página, y creciendo hacia abajo. (Por supuesto, se puede dibujar un diagrama de cualquier manera que uno considere útil.) El diagrama de herencia para **Viento.java** es, por consiguiente:



La conversión de clase derivada a base se mueve hacia *arriba* dentro del diagrama de herencia, por lo que se denomina *conversión hacia arriba.* Esta operación siempre es segura porque se va de un tipo más específico a uno más general. Es decir, la clase derivada es un superconjunto de la clase base. Podría contener más métodos que la clase base, pero debe contener *al menos* los métodos de ésta última. Lo único que puede pasar a la interfaz de clases durante la conversión hacia arriba es que pierda métodos en vez de ganarlos. Ésta es la razón por la que el compilador permite la conversión hacia arriba sin ningún tipo de conversión especial u otras notaciones especiales.

También se puede llevar a cabo lo contrario a la conversión hacia arriba, denominado *conversión hacia abajo*, pero implica el dilema en el que se centra el Capítulo 12.

De nuevo composición frente a herencia

En la programación orientada a objetos, la forma más probable de crear código es simplemente empaquetando juntos datos y métodos en una clase, y usando los objetos de esa clase. También se utilizarán clases existentes para construir nuevas clases con composición. Menos frecuentemente, se usará la herencia. Por tanto, aunque la herencia captura gran parte del énfasis durante el aprendizaje de POO, esto no implica que se deba hacer en todas partes en las que se pueda. Por el contrario, se debe usar de una manera limitada, sólo cuando está claro que es útil. Una de las formas más claras de determinar si se debería usar composición o herencia es preguntar si alguna vez habrá que hacer una conversión hacia arriba desde la nueva clase a la clase base. Si se debe hacer una conversión hacia arriba, entonces la herencia es necesaria, pero si no se necesita, se debería mirar más con detalle si es o no necesaria. El siguiente capítulo (polimorfismo) proporciona una de las razones de más peso para una conversión hacia arriba, pero si uno recuerda preguntar: "¿Necesito una conversión hacia arriba?" obtendrá una buena herramienta para decidir entre la composición y la herencia.

La palabra clave **final**

La palabra clave **final** de Java tiene significados ligeramente diferentes dependiendo del contexto, pero en general dice: "Esto no puede cambiarse". Se podría querer evitar cambios por dos razones: diseño o eficiencia. Dado que estas dos razones son bastante diferentes, es posible utilizar erróneamente la palabra clave **final**.

Las secciones siguientes discuten las tres posibles ubicaciones en las que se puede usar **final**: para datos, métodos y clases.

Para datos

Muchos lenguajes de programación tienen una forma de indicar al compilador que cierta parte de código es "constante". Una constante es útil por varias razones:

1. Puede ser una *constante en tiempo de compilación* que nunca cambiará.
2. Puede ser un valor inicializado en tiempo de ejecución que no se desea que se llegue a cambiar.

En el caso de una constante de tiempo de compilación, el compilador puede "manejar" el valor constante en cualquier cálculo en el que se use; es decir, se puede llevar a cabo el cálculo en tiempo de compilación, eliminando parte de la sobrecarga de tiempo de ejecución. En Java, estos tipos de constantes tienen que ser datos primitivos y se expresan usando la palabra clave **final**. A este tipo de constantes se les debe dar un valor en tiempo de definición.

Un campo que es **estático** y **final** sólo tiene un espacio de almacenamiento que no se puede modificar.

Al usar **final** con referencias a objetos en vez de con datos primitivos, su significado se vuelve algo confuso. Con un dato primitivo, **final** convierte el *valor* en constante, pero con una referencia a un objeto, **final** hace de la *referencia* una constante. Una vez que la referencia se inicializa a un objeto, ésta nunca se puede cambiar para que apunte a otro objeto. Sin embargo, se puede modificar el objeto en sí; Java no proporciona ninguna manera de convertir un objeto arbitrario en una constante. (Sin embargo, se puede escribir la clase, de forma que sus objetos tengan el efecto de ser constantes.) Esta restricción incluye a los arrays, que también son objetos.

He aquí un ejemplo que muestra el funcionamiento de los campos **final**:

```
//: c06:DatosConstantes.java
// El efecto de final en campos.

class Valor {
  int i = 1;
}

public class DatosConstantes {
```

```
  // Pueden ser constantes de tiempo de compilación
  final int i1 = 9;
  static final int VAL_DOS = 99;
  // Típica constante pública:
  public static final int VAL_TRES = 39;
  // No pueden ser constantes en tiempo de compilación:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);

  Valor v1 = new Valor();
  final Valor v2 = new Valor();
  static final Valor v3 = new Valor();
  // Arrays:
  final int[] a = { 1, 2, 3, 4, 5, 6 };

  public void escribir(String id) {
    System.out.println(
      id + ": " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
    DatosConstantes fd1 = new DatosConstantes();
    //! fd1.i1++; // Error: no se puede cambiar el valor
    fd1.v2.i++; // ¡El objeto no es constante!
    fd1.v1 = new Valor(); // OK -- no es final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // ¡El objeto no es una constante!
    //! fd1.v2 = new Valor(); // Error: No se puede
    //! fd1.v3 = new Valor(); // cambiar ahora la referencia
    //! fd1.a = new int[3];

    fd1.escribir("fd1");
    System.out.println("Creando un nuevo DatosConstantes");
    DatosConstantes fd2 = new DatosConstantes();
    fd1.escribir("fd1");
    fd2.escribir("fd2");
  }
} ///:~
```

Dado que **i1** y **VAL_DOS** son datos primitivos **final** con valores de tiempo de compilación, ambos pueden usarse como constantes de tiempo de compilación y su uso no difiere mucho. **VAL_TRES** es la manera más usual en que se verán definidas estas constantes: **pública** de forma que puedan ser utilizadas fuera del paquete, **estática** para hacer énfasis en que sólo hay una, y **final** para indicar que es una constante. Fíjese que los datos primitivo **static final** con valores iniciales constantes (es decir, las constantes de tiempo de compilación) se escriben con mayúsculas por acuerdo,

además de con palabras separadas por guiones bajos (es decir, justo como las constantes de C, que es de donde viene el acuerdo). La diferencia se muestra en la salida de una ejecución:

```
fd1:  i4 = 15; i5 = 9
Creando un nuevo DatosConstante
fd1:  i4 = 15; i5 = 9
fd2:  i4 = 10; i5 = 9
```

Fíjese que los valores de **i4** para **fd1** y **fd2** son únicos, pero el valor de **i5** no ha cambiado al crear el segundo objeto **DatosConstante**. Esto es porque es **estático** y se inicializa una vez en el momento de la carga y no cada vez que se crea un nuevo objeto.

Las variables de **v1** a **v4** demuestran el significado de una referencia **final**. Como se puede ver en **main( )**, justo porque **v2** sea **final,** no significa que no se pueda cambiar su valor. Sin embargo, no se puede reubicar **v2** a un nuevo objeto, precisamente porque es **final**. Eso es lo que **final** significa para una referencia. También se puede ver que es cierto el mismo significado para un array, que no es más que otro tipo de referencia. (No hay forma de convertir en **final** las referencias a array en sí.) Hacer las referencias **final** parece menos útil que hacer **final** a las primitivas.

Constantes blancas

Java permite la creación de *constantes blancas*, que son campos declarados como **final** pero a los que no se da un valor de inicialización. En cualquier caso, se *debe* inicializar una constante blanca antes de utilizarla, y esto lo asegura el propio compilador. Sin embargo, las constantes blancas proporcionan mucha mayor flexibilidad en el uso de la palabra clave **final** puesto que, por ejemplo, un campo **final** incluido en una clase puede ahora ser diferente para cada objeto, y sin embargo, sigue reteniendo su cualidad de inmutable. He aquí un ejemplo:

```
//: c06:CostanteBlanca.java
// Miembros de datos "Constantes blancas".

class Elemento { }

class ConstanteBlanca {
  final int i = 0; // Constante inicializada
  final int j; // Constante blanca
  final Elemento p; // Referencia a constante blanca
  // Las constantes blancas DEBEN inicializarse
  // en el constructor:
  ConstanteBlanca() {
    j = 1; // Inicializar la la constante blanca
    p = new Elemento();
  }
  ConstanteBlanca(int x) {
    j = x; // Inicializar la constante blanca
    p = new Elemento();
```

```
  }
  public static void main(String[] args) {
    ConstanteBlanca bf = new ConstanteBlanca();
  }
} ///:~
```

Es obligatorio hacer asignaciones a **constantes,** bien con una expresión en el momento de definir el campo o en el constructor. De esta forma, se garantiza que el campo **constante** se inicialice siempre antes de ser usado.

Parámetros de valor constante

Java permite hacer parámetros **constantes** declarándolos con la palabra final en la lista de parámetros. Esto significa que dentro del método no se puede cambiar aquello a lo que apunta la referencia al parámetro:

```
//: c06:ParametrosConstante.java
// Utilizando "final" con parámetros de métodos.

class Artilugio {
  public void girar() {}
}

public class ParametrosConstante {
  void con(final Artilugio g) {
    //! g = new Artilugio(); // Ilegal -- g es constante
  }
  void sin(Artilugio g) {
    g = new Artilugio(); // OK -- g no es constante
    g.girar();
  }
  // void f(final int i) { i++; } // No puede cambiar
  // Sólo se puede leer de un tipo de dato primitivo:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    ParametrosConstante bf = new ParametrosConstante();
    bf.sin(null);
    bf.con(null);
  }
} ///:~
```

Fíjese que se puede seguir asignando una referencia **null** a un parámetro **constante** sin que el compilador se dé cuenta, al igual que se puede hacer con un parámetro no **constante**.

Los métodos **f( )** y **g( )** muestran lo que ocurre cuando los parámetros primitivos son **constante**: se puede leer el parámetro pero no se puede cambiar.

Métodos constante

Hay dos razones que justifican los métodos **constante**. La primera es poner un "bloqueo" en el método para evitar que cualquier clase heredada varíe su significado. Esto se hace por razones de diseño cuando uno se quiere asegurar de que se mantenga el comportamiento del método durante la herencia, evitando que sea sobreescrito.

La segunda razón para los métodos **constante** es la eficiencia. Si se puede hacer un método **constante** se está permitiendo al compilador convertir cualquier llamada a ese método en llamadas *rápidas*. Cuando el compilador ve una llamada a un método **constante** puede (a su discreción) saltar el modo habitual de insertar código para llevar a cabo el mecanismo de invocación al método (meter los argumentos en la pila, saltar al código del método y ejecutarlo, volver al punto del salto y eliminar los parámetros de la pila, y manipular el valor de retorno) o, en vez de ello, reemplazar la llamada al método con una copia del código que, de hecho, se encuentra en el cuerpo del método. Esto elimina la sobrecarga de la llamada al método. Por supuesto, si el método es grande, el código comienza a aumentar de tamaño, y probablemente no se aprecien ganancias de rendimiento en la sustitución, puesto que cualquier mejora se verá disminuida por la cantidad de tiempo invertido dentro del método. Está implícito el que el compilador de Java sea capaz de detectar estas situaciones, y elegir sabiamente. Sin embargo, es mejor no confiar en que el compilador sea capaz de hacer esto siempre bien, y hacer un método **constante** sólo si es lo suficientemente pequeño o se desea evitar su modificación explícitamente.

constante y privado

Cualquier método **privado** de una clase es implícitamente **constante**. Dado que no se puede acceder a un método **privado**, no se puede modificar (incluso aunque el compilador no dé un mensaje de error si se intenta modificar, no se habrá modificado el método, sino que se habrá creado uno nuevo). Se puede añadir el modificador **final** a un método **privado** pero esto no da al método ningún significado extra.

Este aspecto puede causar confusión, porque si se desea modificar un método **privado** (que es implícitamente **constante**) parece funcionar:

```
//: c06:AparienciaModificacionConstante.java
// Sólo parece que se puede modificar
// un método privado o privado constante.

class ConConstantes {
  // Idéntico a únicamente "privado":
  private final void f() {
    System.out.println("ConConstantes.f()");
  }
  // También automáticamente "constante":
  private void g() {
    System.out.println("ConConstantes.g()");
  }
}
```

```
class ModificacionPrivado extends ConConstante {
  private final void f() {
    System.out.println("ModificacionPrivado.f()");
  }
  private void g() {
    System.out.println("ModificacionPrivado.g()");
  }
}

class ModificacionPrivado2
  extends ModificacionPrivado {
  public final void f() {
    System.out.println("ModificacionPrivado2.f()");
  }
  public void g() {
    System.out.println("ModificacionPrivado2.g()");
  }
}

public class AparienciaModificacionConstante {
  public static void main(String[] args) {
    ModificacionPrivado2 op2 =
      new ModificacionPrivado2();
    op2.f();
    op2.g();
    // Se puede hacer conversión hacia arriba:
    ModificacionPrivado op = op2;
    // Pero no se puede invocar a los métodos:
    //! op.f();
    //! op.g();
    // Lo mismo que aquí:
    ConCostantes wf = op2;
    //! wf.f();
    //! wf.g();
  }
} ///:~
```

La "modificación" sólo puede darse si algo es parte de la interfaz de la clase base. Es decir, uno debe ser capaz de hacer conversión hacia arriba de un objeto a su tipo base e invocar al mismo método (la esencia de esto se verá más clara en el siguiente capítulo). Si un método es **privado**, no es parte de la interfaz de la clase base. Es simplemente algún código oculto dentro de la clase, y simplemente tiene ese nombre, pero si se crea un método **público**, **protegido** o "amistoso" en la clase derivada, no hay ninguna conexión con el método que pudiese llegar a tener ese nombre en la clase base. Dado que un método **privado** es inalcanzable y a efectos invisible, no influye en nada más que en la organización del código de la clase para la que se definió.

Clases constantes

Cuando se dice que una clase entera es **constante** (precediendo su definición de la palabra clave **final**) se establece que no se desea heredar de esta clase o permitir a nadie más que lo haga. En otras palabras, por alguna razón el diseño de la clase es tal que nunca hay una necesidad de hacer cambios, o por razones de seguridad no se desea la generación de subclases. De manera alternativa, se pueden estar tratando aspectos de eficiencia, y hay que asegurarse de que cualquier actividad involucrada con objetos de esta clase sea lo más eficiente posible.

```
//: c06:Jurasico.java
// Convirtiendo una clase entera en final.

class CerebroPequenio {}

final class Dinosaurio {
  int i = 7;
  int j = 1;
  CerebroPequenio x = new CerebroPequenio();
  void f() {}
}

//! class SerEvolucionado extends Dinosaurio {}
// error: No pueda heredar de la clase constante 'Dinosaurio'

public class Jurasico {
  public static void main(String[] args) {
    Dinosaurio n = new Dinosaurio();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

Fíjese que los atributos pueden ser **constantes** o no, como se desee. Las mismas reglas se aplican a los atributos independientemente de si la clase se ha definido como **constante**. Definiendo la clase como **constante** simplemente evita la herencia —nada más. Sin embargo, dado que evita la herencia, todos los métodos de una clase **constante** son implícitamente **constante**, puesto que no hay manera de modificarlos. Por tanto, el compilador tiene las mismas opciones de eficiencia como tiene si se declara un método explícitamente **constante**.

Se puede añadir el especificador **constante** a un método en una clase **constante**, pero esto no añade ningún significado.

Precaución con constantes

Puede parecer sensato hacer un método **constante** mientras se está diseñando una clase. Uno podría sentir que la eficiencia es muy importante al usar la clase y que nadie podría posiblemente desear modificar estos métodos de ninguna manera. En ocasiones esto es cierto.

Pero hay que ser cuidadoso con las suposiciones. En general, es difícil anticipar cómo se reutilizará una clase, especialmente en el caso de clases de propósito general. Si se define un método como **constante** se podría evitar la posibilidad de reutilizar la clase a través de la herencia en otros proyectos de otros programadores simplemente porque su uso fuera inimaginable.

La biblioteca estándar de Java es un buen ejemplo de esto. En particular, la clase **Vector** de Java 1.0/1.1 se usaba comúnmente y podría haber sido incluso más útil si, en aras de la eficiencia, no se hubieran hecho **constante** todos sus métodos. Es fácil de concebir que se podría desear heredar y superponer partiendo de una clase tan fundamentalmente útil, pero de alguna manera, los diseñadores decidieron que esto no era adecuado. Esto es irónico por dos razones. La primera, que la clase **Stack** hereda de **Vector**, lo que significa que un **Stack** *es* un **Vector**, lo que no es verdaderamente cierto desde el punto de vista lógico. Segundo, muchos de los métodos más importantes de **Vector**, como **addElement( )** y **elementAt( )** están sincronizados (**synchronized**). Como se verá en el Capítulo 14, esto incurre en una sobrecarga considerable que probablemente elimine cualquier ganancia proporcionada por **final**. Esto da credibilidad a la teoría de que los programadores suelen ser normalmente malos a la hora de adivinar dónde deberían intentarse las optimizaciones. Es muy perjudicial que haya un diseño tan poco refinado en una biblioteca con la que todos debemos trabajar. (Afortunadamente, la biblioteca de Java 2 reemplaza **Vector** por **ArrayList**, que se comporta mucho más correctamente. Desgraciadamente, se sigue escribiendo mucho código nuevo que usa la biblioteca antigua.)

También es interesante tener en cuenta que **Hashtable**, otra clase de biblioteca estándar importante, *no* tiene ningún método **constante**. Como se mencionó en alguna otra parte de este libro, es bastante obvio que algunas clases se diseñaron por unas personas y otras por personas completamente distintas. (Se verá que los nombres de método de **Hashtable** son mucho más breves que los de **Vector**, lo cual es otra prueba de esta afirmación.) Este es precisamente el tipo de aspecto que *no* debería ser obvio a los usuarios de una biblioteca de clases. Cuando los elementos son inconsistentes, simplemente el usuario final tendrá que trabajar más. Otra alabanza más al valor del diseño y de los ensayos de código. (Fíjese que la biblioteca de Java 2 reemplaza **Hashtable** por **HashMap**.)

Carga de clases e inicialización

En lenguajes más tradicionales, los programas se cargan de una vez como parte del proceso de arranque. Éste va seguido de la inicialización y posteriormente comienza el programa. El proceso de inicialización en estos lenguajes debe controlarse cuidadosamente de forma que el orden de inicialización de los datos **estáticos** no cause problemas. C++, por ejemplo, tiene problemas si uno de los datos **estáticos** espera que otro dato **estático** sea válido antes de haber inicializado el segundo.

Java no tiene este problema porque sigue un enfoque diferente en la carga. Dado que todo en Java es un objeto, muchas actividades se simplifican, y ésta es una de ellas. Como se aprenderá más en profundidad en el siguiente capítulo, el código compilado de cada clase existe en su propio archivo separado. El archivo no se carga hasta que se necesita el código. En general, se puede decir que "El código de las clases se carga en el momento de su primer uso". Esto no ocurre generalmente hasta que se construye el primer objeto de esa clase, pero también se da una carga cuando se accede a un dato o método **estático**.

El momento del primer uso es también donde se da la inicialización **estática**. Todos los objetos **estáticos** y el bloque de código **estático** se inicializarán en orden textual (es decir, el orden en que se han escrito en la definición de la clase) en el momento de la carga. Los datos **estáticos**, por supuesto, se inicializan únicamente una vez.

Inicialización con herencia

Ayuda a echar un vistazo a todo el proceso de inicialización, incluyendo la herencia, para conseguir una idea global de lo que ocurre. Considérese el siguiente código:

```
//: c06:Escarabajo.java
// El proceso de inicialización completo.

class Insecto {
  int i = 9;
  int j;
  Insecto() {
    visualizar("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    visualizar("static Insecto.x1 inicializado");
  static int visualizar(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Escarabajo extends Insecto {
  int k = visualizar("Escarabajo.k inicializado");
  Escarabajo() {
    visualizar("k = " + k);
    visualizar("j = " + j);
  }
  static int x2 =
    visualizar("static escarabajo.x2 inicializado");
  public static void main(String[] args) {
```

```
        visualizar("Constructor de Escarabajos ");
        Escarabajo b = new Escarabajo();
    }
} ///:~
```

La salida de este programa es:

```
static Insecto.x1 inicializado
static Escarabajo.x2 inicializado
Constructor de Escarabajos i = 9, j = 0
Escarabajo.k inicializado
k = 47
j = 39
```

Lo primero que ocurre al ejecutar **Escarabajo** bajo Java es que se intenta acceder a **Escarabajo.main( )** (un método **estático**), de forma que el cargador sale a buscar el código compilado de la clase **Escarabajo** (que resulta estar en un fichero denominado **Escarabajo.class**). En el proceso de su carga, el cargador se da cuenta de que tiene una clase base (que es lo que indica la palabra clave **extends**), y por consiguiente, la carga. Esto ocurrirá tanto si se hace como si no un objeto de esa clase. (Intente comentar la creación del objeto si se desea demostrar esto.)

Si la clase base tiene una clase base, las segunda clase base se cargaría también, y así sucesivamente. Posteriormente, se lleva a cabo la inicialización **estática** de la clase base raíz (en este caso **Insecto**), y posteriormente la siguiente clase derivada, y así sucesivamente. Esto es importante porque la inicialización estática de la clase derivada podría depender de que se inicialice adecuadamente el miembro de la clase base.

En este momento, las clases necesarias ya han sido cargadas de forma que se puede crear el objeto. Primero, se ponen a sus valores por defecto todos los datos primitivos de este objeto, y las referencias a objetos se ponen a **null** —esto ocurre en un solo paso poniendo la memoria del objeto a ceros binarios. Después se invoca al constructor de la clase base. En este caso, la llamada es automática, pero también se puede especificar la llamada al constructor de la clase base (como la primera operación en el constructor de **Escarabajo( )**) utilizando **super**. La construcción de la clase base sigue el mismo proceso en el mismo orden, como el constructor de la clase derivada. Una vez que acaba el constructor de la clase base se inicializan las variables de instancia en orden textual. Finalmente se ejecuta el resto del cuerpo del constructor.

Resumen

Tanto la herencia como la composición, permiten crear un nuevo tipo a partir de tipos ya existentes. Generalmente, sin embargo, se usa la composición para reutilizar tipos ya existentes como parte de la implementación subyacente del nuevo tipo, y la herencia cuando se desee reutilizar la interfaz. Dado que la clase derivada tiene la interfaz de la clase base, se le puede hacer una *conversión hacia arriba* hasta la clase base, lo que es crítico para el polimorfismo, como se verá en el siguiente capítulo.

A pesar del gran énfasis que la programación orientada a objetos pone en la herencia, al empezar un diseño debería generalmente preferirse la composición durante el primer corte, y la herencia sólo cuando sea claramente necesaria. La composición tiende a ser más flexible. Además, al utilizar la propiedad añadida de la herencia con un tipo miembro, se puede cambiar el tipo exacto y, por tanto, el comportamiento de aquellos objetos miembro en tiempo de ejecución. Por consiguiente, se puede cambiar el comportamiento del objeto compuesto en tiempo de ejecución.

Aunque la reutilización de código mediante la composición y la herencia es útil para el desarrollo rápido de proyectos, generalmente se deseará rediseñar la jerarquía de clases antes de permitir a otros programadores llegar a ser dependientes de ésta. La meta es una jerarquía en la que cada clase tenga un uso específico y no sea demasiado grande (agrupando tanta funcionalidad sería demasiado difícil de manejar) ni demasiado pequeño (no se podría usar por sí mismo o sin añadirle funcionalidad).

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Crear dos clases, **A** y **B**, con constructores por defecto (listas de parámetros vacías) que se anuncien a sí mismas. Heredar una nueva clase **C** a partir de **A**, y crear un miembro de la clase **B** dentro de **C**. No crear un constructor para **C**. Crear un objeto de la clase **C** y observar los resultados.

2. Modificar el Ejercicio 1 de forma que **A** y **B** tengan constructores con parámetros en vez de constructores por defecto. Escribir un constructor para **C** y llevar a cabo toda la inicialización dentro del constructor **C**.

3. Crear una clase simple. Dentro de una segunda clase, definir un campo para un objeto de la primera clase. Utilizar inicialización perezosa para instanciar este objeto.

4. Heredar una nueva clase de la clase **Detergente**. Superponer **frotar( )** y añadir un nuevo método denominado **esterilizar( )**.

5. Tomar el archivo **Animacion.java** y comentar el constructor de la clase **Animación**. Explicar qué ocurre.

6. Tomar el archivo **Ajedrez.java** y comentar el constructor de la clase **Ajedrez**. Explicar qué ocurre.

7. Probar que se crean constructores por defecto por parte del compilador.

8. Probar que los constructores de una clase base (a) siempre son invocados y, (b) se invocan antes que los constructores de la clase derivada.

9. Crear una clase base con sólo un constructor distinto del constructor por defecto, y una clase derivada que tenga, tanto un constructor por defecto, como uno que no lo sea. En los constructores de la clase derivada, invocar al de la clase base.

10. Crear una clase llamada **Raíz** que contenga una instancia de cada clase (que también se deben crear) denominadas **Componente1**, **Componente2**, y **Componente3**. Derivar una clase **Tallo** a partir de **Raíz** que también contenga una instancia de cada "componente". Todas las clases deberían tener constructores por defecto que impriman un mensaje relativo a ellas.

11. Modificar el Ejercicio 10 de forma que cada clase sólo tenga un constructor que no sea por defecto.

12. Añadir una jerarquía correcta de métodos **limpiar( )** a todas las clases del Ejercicio 11.

13. Crear una clase con un método sobrecargado tres veces. Heredar una nueva clase, añadir una nueva sobrecarga del método y mostrar que los cuatro métodos están disponibles para la clase derivada.

14. En **Coche.java** añadir un método **revisar( )** a **Motor** e invocar a este método en el método **main( )**.

15. Crear una clase dentro de un paquete. La clase debería contener un método **protegido**. Fuera del paquete, intentar invocar al método **protegido** y explicar los resultados. Ahora heredar de la clase e invocar al método **protegido** desde dentro del método de la clase derivada.

16. Crear una clase llamada **Anfibio**. Desde ésta, heredar una clase llamada **Rana**. Poner métodos apropiados en la clase base. En el método **main( )**, crear una **Rana** y hacer una conversión hacia **Anfibio**. Demostrar que todos los métodos siguen funcionando.

17. Modificar el Ejercicio 16 de forma que **Rana** superponga las definiciones de métodos de la clase base (proporciona nuevas definiciones usando los mismos nombres de método). Fijarse en lo que ocurre en el método **main( )**.

18. Crear una clase con un campo **estático constante** y un campo **constante**, y demostrar la diferencia entre los dos.

19. Crear una clase con una referencia constante **blanca** a un objeto. Llevar a cabo la inicialización de la constante *blanca* **final** dentro del método (no en el constructor) justo antes de usarlo. Demostrar que debe inicializarse la **constante** antes de usarlos, y que no puede cambiarse una vez inicializada.

20. Crear una clase con un método **constante**. Heredar desde esa clase e intentar superponer ese método.

21. Crear una clase **constante** e intentar heredar de ella.

22. Probar que la carga de clases sólo se da una vez. Probar que la carga puede ser causada, bien por la creación de la primera instancia de esa clase, o por el acceso a un miembro **estático**.

23. En **Escarabajo.java**, heredar un tipo específico de escarabajo de la clase **Escarabajo**, siguiendo el mismo formato que el de la clase existente. Hacer un seguimiento y explicar la salida.

7: Polimorfismo

El polimorfismo es la tercera característica esencial de los lenguajes de programación orientados a objetos, después de la abstracción de datos y la herencia.

Proporciona otra dimensión de separación de la interfaz de la implementación, separa el *qué* del *cómo*. El polimorfismo permite una organización de código y una legibilidad del mismo mejorada, además de la creación de programas *ampliables* que pueden "crecer", no sólo durante la creación original del proyecto sino también cuando se deseen añadir nuevas características.

La encapsulación crea nuevos tipos de datos mediante la combinación de características y comportamientos. La ocultación de la implementación separa la interfaz de la implementación haciendo los detalles **privados**. Este tipo de organización mecánica tiene bastante sentido para alguien con un trasfondo procedural de programación. Pero el polimorfismo tiene que ver con la separación en términos de *tipos*. En el capítulo anterior se vio como la herencia permite el tratamiento de un objeto como si fuera de sus propio tipo *o* del tipo base. Esta característica es crítica porque permite que varios tipos (derivados de un mismo tipo base) sean tratados como si fueran uno sólo, y un único fragmento de código se puede ejecutar de igual forma en todos los tipos diferentes. La llamada a un método polimórfico permite que un tipo exprese su distinción de otro tipo similar, puesto que ambos se derivan del mismo tipo base. Esta distinción se expresa a través de diferencias en comportamiento de los métodos a los que se puede invocar a través de la clase base.

En este capítulo, se aprenderá lo relacionado con el polimorfismo (llamado también *reubicación dinámica*, *reubicación tardía* o *reubicación en tiempo de ejecución*) partiendo de la base, con ejemplos simples que prescinden de todo, menos del comportamiento polimórfico del programa.

De nuevo la conversión hacia arriba

En el Capítulo 6 se vio cómo un objeto puede usarse con su propio tipo o como un objeto de su tipo base. Tomar una referencia a un objeto y tratarla como una referencia a su clase base se denomina *conversión hacia arriba*, debido a la forma en que se dibujan los árboles de herencia, en los que la clase base se coloca siempre en la parte superior.

También se vio que surge un problema, como se aprecia en:

```
//: c07:musica:Musica.java
// Herencia y conversión hacia arriba.

class Nota {
  private int valor;
  private Nota(int val) { valor = val; }
  public static final Nota
    DO_MAYOR = new Nota(0),
    DO_SOSTENIDO  = new Nota(1),
```

```
      SI_BEMOL    = new Nota(2);
} // Etc.

class Instrumento {
   public void tocar(Nota n) {
      System.out.println("Instrumento.tocar()");
   }
}

// Los objetos de viento son instrumentos
// dado que tienen la misma interfaz:
class Viento extends Instrumento {
   // Redefinir el metodo interfaz:
   public void tocar(Nota n) {
      System.out.println("Viento.tocar()");
   }
}

public class Musica {
   public static void afinar(Instrumento i) {
      // ...
      i.tocar(Nota.DO_SOSTENIDO);
   }
   public static void main(String[] args) {
      Viento flauta = new Viento();
      afinar(flauta); // Conversión hacia arriba
   }
} ///:~
```

El método **Musica.afinar( )** acepta una referencia a **Instrumento**, pero también cualquier cosa que se derive de **Instrumento**. En el método **main( )**, se puede ver que ocurre esto pues se pasa una referencia **Viento** a **afinar( )**, sin que sea necesaria ninguna conversión. Esto es aceptable; la interfaz de **Instrumento** debe existir en **Viento,** puesto que **Viento** se hereda de **Instrumento**. Hacer una conversión hacia arriba de **Viento** a **Instrumento** puede "estrechar" esa interfaz, pero no puede reducirlo a nada menos de lo contenido en la interfaz de **Instrumento**.

Olvidando el tipo de objeto

Este programa podría parecer extraño. ¿Por qué debería alguien *olvidar* intencionadamente el tipo de objeto? Esto es lo que ocurre cuando se hace conversión hacia arriba, y parece que podría ser mucho más directo si **afinar( )** simplemente tomara una referencia **Viento** como argumento. Esto presenta un punto esencial: si se hiciera esto se necesitaría escribir un nuevo método **afinar( )** para cada tipo de **Instrumento** del sistema. Supóngase que se sigue este razonamiento y se añaden los instrumentos de **Cuerda** y **Metal**:

```
//: c07:musica2:Musica2.java
// Sobrecarga en vez de conversión hacia arriba.

class Nota {
  private int valor;
  private Nota(int val) { valor = val; }
  public static final Nota
    DO-MAYOR = new Nota(0),
    DO-SOSTENIDO = new Nota(1),
    SI-BEMOL = new Nota(2);
} // Etc.

class Instrumento {
  public void tocar(Nota n) {
    System.out.println("Instrumento.tocar()");
  }
}

class Viento extends Instrumento {
  public void tocar(Nota n) {
    System.out.println("Viento.tocar()");
  }
}

class Cuerda extends Instrumento {
  public void tocar(Nota n) {
    System.out.println("Cuerda.tocar()");
  }
}

class Metal extends Instrumento {
  public void tocar(Nota n) {
    System.out.println("Metal.tocar()");
  }
}

public class Musica2 {
  public static void afinar(Viento i) {
    i.tocar(Nota.DO-MAYOR);
  }
  public static void afinar(Cuerda i) {
    i.tocar(Nota.DO-MAYOR);
  }
  public static void afinar(Metal i) {
    i.tocar(Nota.DO-MAYOR);
```

```
    }
    public static void main(String[] args) {
        Viento flauta = new Viento();
        Cuerda violin = new Cuerda();
        Metal trompeta = new Metal();
        afinar(flauta); // Sin conversión hacia arriba
        afinar(violin);
        afinar(trompeta);
    }
} ///:~
```

Esto funciona, pero hay un inconveniente: se deben escribir métodos específicos de cada tipo para cada clase **Instrumento** que se añada. En primer lugar esto significa más programación, pero también quiere decir que si se desea añadir un método nuevo como **afinar( )** o un nuevo tipo de **Instrumento**, se tiene mucho trabajo por delante. Añadiendo el hecho de que el compilador no emitirá ningún mensaje de error si se olvida sobrecargar alguno de los métodos, el hecho de trabajar con tipos podría convertirse en inmanejable.

¿No sería muchísimo mejor si simplemente se pudiera escribir un único método que tomara como parámetro la clase base, y no cualquiera de las clases específicas derivadas? Es decir, ¿no sería genial que uno se pudiera olvidar de que hay clases derivadas, y escribir un código que sólo tratara con la clase base?

Esto es exactamente lo que permite hacer el polimorfismo. Sin embargo, la mayoría de programadores que provienen de lenguajes procedurales, tienen problemas para entender el funcionamiento de esta caracterítica.

El cambio

La dificultad con **Musica.java** se puede ver ejecutando el programa. La salida es **Viento.tocar( )** Ésta es, ciertamente, la salida deseada, pero no parece tener sentido que funcione de esa forma. Obsérvese el método **afinar( )**:

```
    public static void afinar(Instrumento i) {
        // ...
        i.tocar(Nota.DO-MAYOR);
    }
```

Recibe una referencia a **Instrumento**. Por tanto, ¿cómo puede el compilador saber que esta referencia a **Instrumento** apunta a **Viento** en este caso, y no a **Cuerda** o **Metal**? El compilador de hecho no puede saberlo. Para lograr un entendimiento más profundo de este aspecto, es útil echar un vistazo al tema de la *ligadura*.

La ligadura en las llamadas a métodos

La conexión de una llamada a un método se denomina *ligadura*. Cuando se lleva a cabo la *ligadura* antes de ejecutar el programa (por parte del compilador y el montador, cuando lo hay) se denomina *ligadura temprana*. Puede que este término parezca extraño pues nunca ha sido una opción con los lenguajes procedurales. Los compiladores de C tienen un único modo de invocar a un método utilizando la ligadura temprana.

La parte confusa del programa de arriba no se resuelve fácilmente con la *ligadura temprana* pues el compilador no puede saber el método correcto a invocar cuando sólo tiene una referencia a un **Instrumento**.

La solución es la *ligadura tardía*, que implica que la correspondencia se da en tiempo de ejecución, basándose en el tipo de objeto. La *ligadura tardía* se denomina también *dinámica* o *en tiempo de ejecución*. Cuando un lenguaje implementa la ligadura tardía, debe haber algún mecanismo para determinar el tipo de objeto en tiempo de ejecución e invocar al método adecuado. Es decir, el compilador sigue sin saber el tipo de objeto, pero el mecanismo de llamada a métodos averigua e invoca al cuerpo de método correcto. El mecanismo de *la ligadura tardía* varía de un lenguaje a otro, pero se puede imaginar que es necesario instalar algún tipo de información en los objetos.

Toda ligadura de métodos en Java se basa en la ligadura tardía a menos que se haya declarado un método como **constante**. Esto significa que ordinariamente no es necesario tomar decisiones sobre si se dará la ligadura tardía, sino que esta decisión se tomará automáticamente.

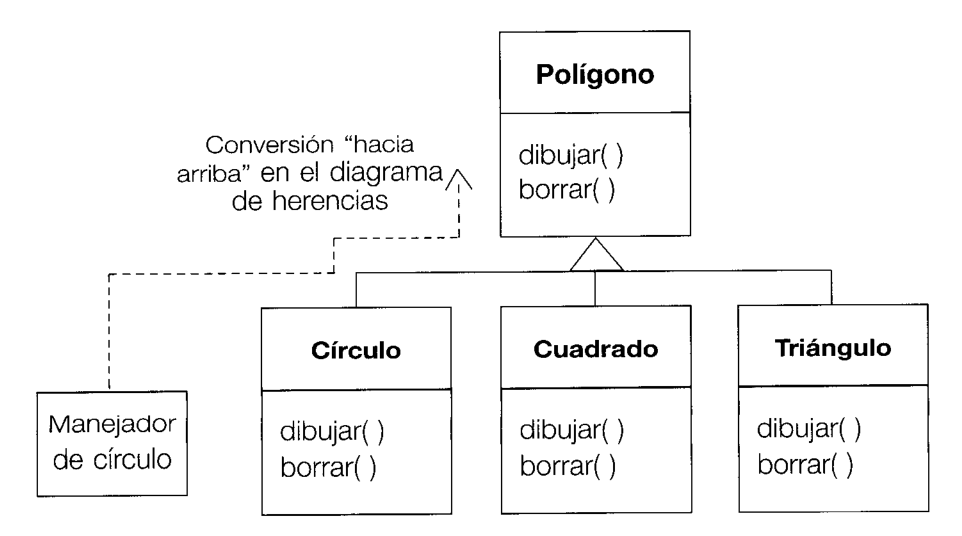
¿Por qué declarar un método como **constante**? Como se comentó en el capítulo anterior, evita que nadie superponga el método. Todavía más importante, "desactiva" ligadura dinámica, o mejor, es que, le dice al compilador que este tipo de ligadura no es necesaria. Esto permite al compilador generar código ligeramente más eficiente para llamadas a métodos **constantes**. Sin embargo, en la mayoría de los casos no se obtendrá ninguna mejora global de rendimiento del programa, por lo que es mejor usar métodos **constantes** únicamente como una decisión de diseño, y no para intentar mejorar el rendimiento.

Produciendo el comportamiento adecuado

Una vez que se sabe que toda la ligadura de métodos en Java se da de forma polimórfica a través de ligadura tardía, se puede escribir código que trate la clase base y saber que todas las clases derivadas funcionarán correctamente al hacer uso de ese mismo código. Dicho de otra forma, se "envía un mensaje a un objeto y se deja que éste averigüe la opción correcta a realizar".

El ejemplo clásico de POO es el ejemplo de los "polígonos". Éste se usa frecuentemente porque es fácil de visualizar, pero desgraciadamente puede confundir a los programadores novatos, haciéndoles pensar que la POO sólo se usa en programación de gráficos, y esto no es cierto.

El ejemplo de los polígonos tiene una clase base denominada **Polígono** y varios tipos derivados: **Círculo**, **Cuadrado** y **Triángulo**, etc. La razón por la que el ejemplo funciona tan bien es porque se puede decir sin problema "un círculo es un tipo de polígono" y se entiende. El diagrama de herencia muestra las relaciones:



La conversión hacia arriba podría darse en una sentencia tan simple como:

```
Poligono s = new Circulo();
```

Aquí, se crea un objeto **Círculo** y la referencia resultante se asigna directamente a un **Polígono**, lo que podría parecer un error (asignar un tipo a otro); y sin embargo, está bien porque un **Círculo** *es un* **Polígono** por herencia. Por tanto, el compilador se muestra de acuerdo con la sentencia y no muestra ningún mensaje de error.

Supóngase que se invoca a uno de los métodos de la clase base (que han sido superpuestos en clases derivadas):

```
s.dibujar();
```

De nuevo, se podría esperar que se invoque al método **dibujar( )** de **Polígono** porque se trata, después de todo, de una referencia a **Polígono** —por tanto, ¿cómo podría el compilador saber que tiene que hacer otra cosa? Y sin embargo, se invoca al **Círculo.dibujar( )** correcto debido a la ligadura tardía (polimorfismo).

El ejemplo siguiente hace lo propio de una manera ligeramente distinta:

```
//: c07:Poligonos.java
// Polimorfismo en Java.

class Poligono {
  void dibujar() {}
  void borrar() {}
}

class Circulo extends Poligono {
  void dibujar() {
    System.out.println("Circulo.dibujar()");
```

```
  }
  void borrar() {
    System.out.println("Circulo.borrar()");
  }
}

class Cuadrado extends Poligono {
  void dibujar() {
    System.out.println("Cuadrado.dibujar()");
  }
  void borrar() {
    System.out.println("Cuadrado.borrar()");
  }
}

class Triangulo extends Poligono {
  void dibujar() {
    System.out.println("Triangulo.dibujar()");
  }
  void borrar() {
    System.out.println("Triangulo.borrar()");
  }
}

public class Poligonos {
  public static PoligonoAleatorio () {
    switch((int)(Math.random() * 3)) {
      default:
      case 0: return new Circulo();
      case 1: return new Cuadrado();
      case 2: return new Triangulo();
    }
  }
  public static void main(String[] args) {
    Poligono[] s = new Poligono[9];
    // Rellenar el array con Polígonos:
    for(int i = 0; i < s.length; i++)
      s[i] = PoligonoAleatorio();
    // Hacer llamadas a métodos polimórficos:
    for(int i = 0; i < s.length; i++)
      s[i].dibujar();
  }
} ///:~
```

La clase base **Polígono** establece la interfaz común a cualquier cosa heredada de **Polígono** —es decir, se pueden borrar y dibujar todos los polígonos. La clase derivada superpone estas definiciones para proporcionar un comportamiento único para cada tipo específico de polígono.

La clase principal **Polígonos** contiene un método **estático** llamado **poligonoAleatorio( )** que produce una referencia a un objeto **Polígonos** seleccionado al azar cada vez que se le invoca. Fíjese que se realiza una conversión hacia arriba en cada sentencia **return** que toma una referencia a un **Círculo**, **Cuadrado** o **Triángulo,** y lo envía fuera del método con tipo de retorno **Polígonos**. Así, al invocar a este método no se tendrá la opción de ver de qué tipo específico es el valor devuelto, dado que siempre se obtendrá simplemente una referencia a **Polígono**.

El método **main( )** contiene un array de referencias **Polígono** rellenadas mediante llamadas a **poligonoAleatorio( )**. En este punto se sabe que se tienen objetos de tipo **Polígono**, pero no se sabe nada sobre nada más específico que eso (y tampoco el compilador). Sin embargo, cuando se recorre este array y se invoca al método **dibujar( )** para cada uno de sus objetos, mágicamente se da el comportamiento correcto específico de cada tipo, como se puede ver en un ejemplo de salida:

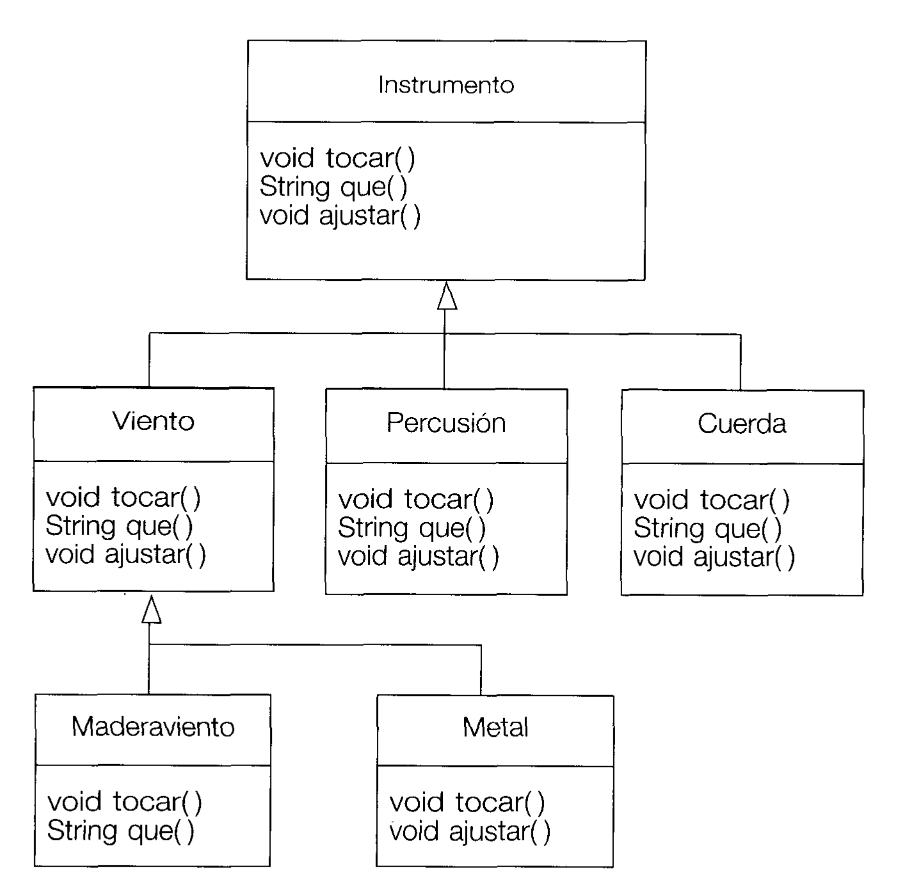
```
Circulo.dibujar()
Triangulo.dibujar()
Circulo.dibujar()
Circulo.dibujar()
Circulo.dibujar()
Cuadrado.dibujar()
Triangulo.dibujar()
Cuadrado.dibujar()
Cuadrado.dibujar()
```

Por supuesto, dado que los polígonos se eligen cada vez al azar, cada ejecución tiene resultados distintos. El motivo de elegir los polígonos al azar es abrirse paso por la idea de que el compilador no puede tener ningún conocimiento que le permita hacer las llamadas correctas en tiempo de compilación. Todas las llamadas a **dibujar( )** se hacen mediante ligadura dinámica.

Extensibilidad

Ahora, volvamos al ejemplo de los instrumentos musicales. Debido al polimorfismo, se pueden añadir al sistema tantos tipos como se desee sin cambiar el método **afinar( )**. En un programa POO bien diseñado, la mayoría de métodos deberían seguir el modelo de **afinar( )** y comunicarse sólo con la interfaz de la clase base. Un programa así es *extensible* porque se puede añadir nueva funcionalidad heredando nuevos tipos de datos de la clase base común. Los métodos que manipulan la interfaz de la clase base no necesitarán ningún cambio si se desea acomodarlos a las nuevas clases.

Considérese qué ocurre si se toma el ejemplo de los instrumentos y se añaden nuevos métodos a la clase base y varias clases nuevas. He aquí el diagrama:



Todas estas nuevas clases funcionan correctamente con el viejo método **afinar( )** sin tocarlo. Incluso si **afinar( )** se encuentra en un archivo separado y se añaden métodos de la interfaz de **Instrumento**, **afinar( )** funciona correctamente sin tener que volver a compilarlo. He aquí una implementación del diagrama de arriba:

```
//: c07:musica3:Musica3.java
// Un programa extensible.
import java.util.*;

class Instrumento {
  public void tocar() {
    System.out.println("Instrumento.tocar()");
  }
  public String que() {
    return "Instrumento";
  }
  public void ajustar() {}
}

class Viento extends Instrumento {
```

```
  public void tocar() {
    System.out.println("Viento.tocar()");
  }
  public String que() { return "Viento"; }
  public void ajustar() {}
}

class Percusion extends Instrumento {
  public void tocar() {
    System.out.println("Percusion.tocar()");
  }
  public String que() { return "Percusion"; }
  public void ajustar() {}
}

class Cuerda extends Instrumento {
  public void tocar() {
    System.out.println("Cuerda.tocar()");
  }
  public String que() { return "Cuerda"; }
  public void ajustar() {}
}

class Metal extends Viento {
  public void tocar() {
    System.out.println("Metal.tocar()");
  }
  public void ajustar() {
    System.out.println("Metal.ajustar()");
  }
}

class Maderaviento extends Viento {
  public void tocar() {
    System.out.println("Maderaviento.tocar()");
  }
  public String que() { return "Maderaviento"; }
}

public class Musica3 {
  // No le importa el tipo por lo que los nuevos tipos
  // que se aniadan al sistema seguirán funcionando bien:
  static void afinar(Instrumento i) {
    // ...
    i.tocar();
```

```
  }
  static void afinarTodo(Instrumento[] e) {
    for(int i = 0; i < e.length; i++)
      afinar(e[i]);
  }
  public static void main(String[] args) {
    Instrumento[] orquesta = new Instrumento[5];
    int i = 0;
    // Conversión hacia arriba durante inserción en el array:
    orquesta[i++] = new Viento();
    orquesta[i++] = new Percusion();
    orquesta[i++] = new Cuerda();
    orquesta[i++] = new Metal();
    orquesta[i++] = new Maderaviento();
    afinarTodo(orquesta);
  }
} ///:~
```

Los nuevos métodos son **que( )**, que devuelve una referencia a una cadena de caracteres con una descripción de la clase, y **ajustar( )**, que proporciona alguna manera de ajustar cada instrumento.

En **main( )**, cuando se coloca algo dentro del array **Instrumento** se puede hacer una conversión hacia arriba automáticamente a **Instrumento**.

Se puede ver que el método **afinar( )** ignora por completo todos los cambios de código que hayan ocurrido alrededor, y sigue funcionando correctamente. Esto es exactamente lo que se supone que proporciona el polimorfismo. Los cambios en el código no causan daño a partes del programa que no deberían verse afectadas. Dicho de otra forma, el polimorfismo es una de las técnicas más importantes que permiten al programador "separar los elementos que cambian de aquellos que permanecen igual".

Superposición frente a sobrecarga

Tomemos un enfoque distinto del primer enfoque de este capítulo. En el programa siguiente, se cambia la interfaz del método **tocar( )** en el proceso de sobrecarga, lo que significa que no se ha *superpuesto* el método, sino que se ha *sobrecargado*. El compilador permite sobrecargar métodos de forma que no haya quejas. Pero el comportamiento no es probablemente lo que se desea. He aquí un ejemplo:

```
//: c07:ErrorViento.java
// Cambiando la interfaz accidentalmente.

class NotaX {
  public static final int
    DO_MAYOR_C = 0, DO_SOSTENIDO = 1, SI_BEMOL = 2;
```

```
}

class InstrumentoX {
  public void tocar(int NotaX) {
    System.out.println("InstrumentoX.tocar()");
  }
}

class VientoX extends InstrumentoX {
  // Cambia la interfaz del método:
  public void tocar(NotaX n) {
    System.out.println("VientoX.tocar(NotaX n)");
  }
}

public class ErrorViento {
  public static void afinar(InstrumentoX i) {
    // ...
    i.tocar(NoteX.DO_MAYOR);
  }
  public static void main(String[] args) {
    VientoX flauta = new VientoX();
    afinar(flauta); // ¡No es el comportamiento deseado!
  }
} ///:~
```

Hay otro aspecto confuso en este caso. En **InstrumentoX**, el método **tocar( )** toma un dato **entero** con el identificador **NotaX**. Es decir, incluso aunque **NotaX** es un nombre de clase, también puede usarse como identificador sin problemas. Pero en **VientoX**, **tocar( )** toma una referencia a **NotaX** que tiene un identificador **n**. (Aunque podría incluso decirse **tocar(NotaX NotaX)** sin que diera error.) Por consiguiente parece que el programador pretendía superponer **tocar( )** pero equivocó los tipos del método. El compilador, sin embargo, asumió que se pretendía una sobrecarga y no una superposición. Fíjese que si se sigue la convención de nombres estándar de Java, el identificador de parámetros sería **notaX** ('n' minúscula), que lo distinguiría del nombre de la clase.

En **afinar**, se envía al **InstrumentoX i** el mensaje **tocar( )**, con uno de los miembros de **NotaX** (**DO_MAYOR**) como parámetro. Dado que **NotaX** contiene definiciones **int**, se invoca a la versión ahora sobrecargada del método **tocar( )**, y dado que ése *no* ha sido superpuesto, se emplea la versión de la clase base.

La salida es:

```
InstrumentoX.tocar()
```

Ciertamente esto no parece ser una llamada a un método polimórfico. Una vez que se entiende lo que está ocurriendo, se puede solventar el problema de manera bastante sencilla, pero imagínese lo

difícil que podría ser encontrar el fallo cuando se encuentre inmerso en un programa de tamaño significativo.

Clases y métodos abstractos

En todos los ejemplos de instrumentos, los métodos de la clase base **Instrumento** eran métodos "falsos". Si se llega a invocar alguna vez a estos métodos daría error. Esto es porque la intención de **Instrumento** es simplemente crear una *interfaz común* para todas las clases que se derivan del mismo.

La única razón para establecer esta interfaz común es que ésta se pueda expresar de manera distinta para cada subtipo diferente. Establece una forma básica, de forma que se puede decir qué tiene en común con todas las clases derivadas.

Otra manera de decir esto es llamar a la clase **Instrumento**, una *clase base abstracta* (o simplemente *clase abstracta*). Se crea una clase abstracta cuando se desea manipular un conjunto de clases a través de una interfaz común. Todos los métodos de clases derivadas que encajen en la declaración de la clase base se invocarán utilizando el mecanismo de ligadura dinámica. (Sin embargo, como se vio en la sección anterior, si el nombre del método es el mismo que en la clase base, pero los parámetros son diferentes, se tiene sobrecarga, lo cual probablemente no es lo que se desea.)

Si se tiene una clase abstracta como **Instrumento**, los objetos de esa clase casi nunca tienen significado. Es decir, **Instrumento** simplemente tiene que expresar la interfaz, y no una implementación particular, de forma que no tiene sentido crear objetos de tipo **Instrumento**, y probablemente se desea evitar que ningún usuario llegue a hacerlo. Esto se puede lograr haciendo que todos los métodos de **Instrumento** muestren mensajes de error, pero de esta forma se retrasa la información hasta tiempo de ejecución, y además es necesaria una comprobación exhaustiva y de confianza por parte del usuario. Siempre es mejor capturar los problemas en tiempo de ejecución.

Java proporciona un mecanismo para hacer esto, denominado el *método abstracto*[1]. Se trata de un método incompleto; tiene sólo declaración faltándole los métodos. La sintaxis para una declaración de método abstracto es:

```
abstract void f();
```

Toda clase que contenga uno o más métodos abstractos, se califica de **abstracto**. (De todos modos, el compilador emite un mensaje de error).

Si una clase abstracta está incompleta, ¿qué se supone que debe hacer el compilador cuando alguien intenta crear un objeto de esa clase? No se puede crear un objeto de una clase abstracta de forma segura, por lo que se obtendrá un mensaje de error del compilador. De esta manera, el compilador asegura la pureza de la clase abstracta, y no hay que preocuparse de usarla mal.

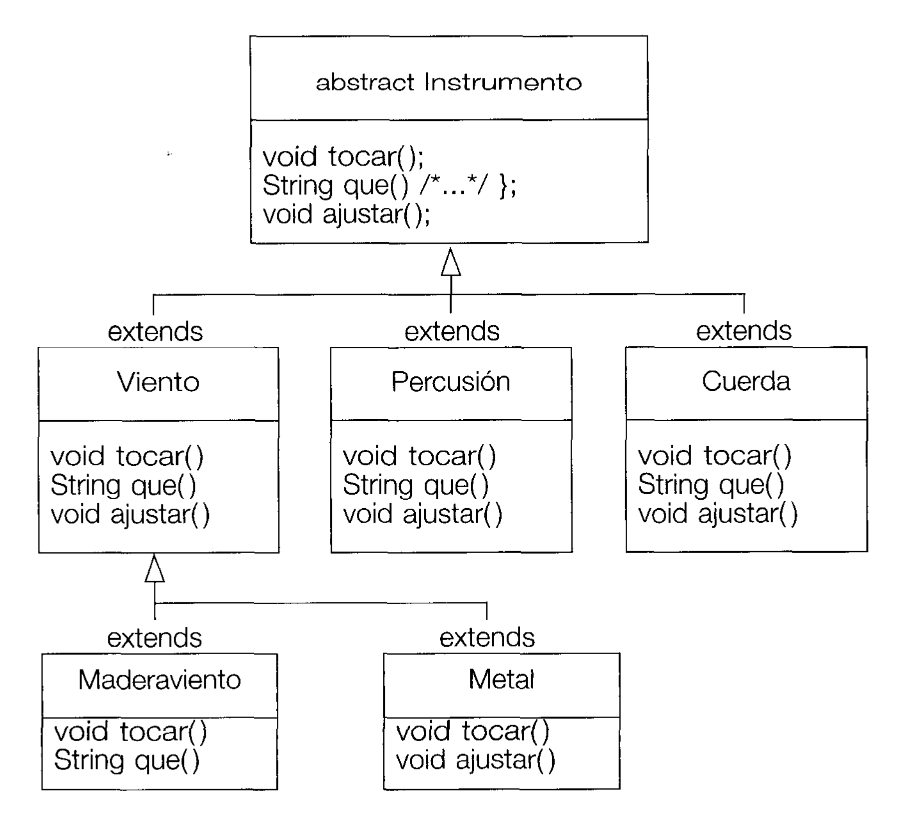
Si se hereda de una clase abstracta y se desea hacer objetos del nuevo tipo, hay que proporcionar definiciones de métodos para todos los métodos que en la clase base eran abstractos. Si no se hace así (y uno puede elegir no hacerlo) entonces la clase derivada también será abstracta y el compilador obligará a calificar *esa* clase con la palabra clave **abstract.**

---

[1] Para los programadores de C++, esto es análogo a la *función virtual pura* de C++.

Es posible crear una clase **abstracta** sin incluir ningún método **abstracto** en ella. Esto es útil cuando se desea una clase en la que no tiene sentido tener métodos **abstractos**, y se desea evitar que existan instancias de esa clase.

La clase **Instrumento** puede convertirse fácilmente en una clase **abstracta**. Sólo serán **abstractos** alguno de los métodos, puesto que hacer una clase **abstracta** no fuerza a hacer **abstractos** todos sus métodos. Quedará del siguiente modo:



He aquí el código de la orquesta modificado para que use clases y métodos **abstractos**:

```
//: c07:musica4:Musica4.java
// Clases y métodos abstractos.
import java.util.*;

abstract class Instrumento {
  int i; // almacenamiento asignado a cada uno
  public abstract void tocar();
  public String que() {
    return "Instrumento";
  }
  public abstract void ajustar();
}
```

```
class Viento extends Instrumento {
   public void tocar() {
      System.out.println("Viento.tocar()");
   }
   public String que() { return "Viento"; }
   public void ajustar() {}
}

class Percusion extends Instrumento {
   public void tocar() {
      System.out.println("Percusion.tocar()");
   }
   public String que() { return "Percusion"; }
   public void ajustar() {}
}

class Cuerda extends Instrumento {
   public void tocar() {
      System.out.println("Cuerda.tocar()");
   }
   public String que() { return "Cuerda"; }
   public void ajustar() {}
}

class Metal extends Viento {
   public void tocar() {
      System.out.println("Metal.tocar()");
   }
   public void ajustar() {
      System.out.println("Metal.ajustar()");
   }
}

class Maderaviento extends Viento {
   public void tocar() {
      System.out.println("Maderaviento.tocar()");
   }
   public String que() { return "Maderaviento"; }
}

public class Musica4 {
   // No le importa el tipo, por lo que los nuevos tipos
   // que se aniadan al sistema seguirán funcionando correctamente:
   static void afinar(Instrumento i) {
```

```
      // ...
      i.tocar();
   }
   static void afinarTodo(Instrumento[] e) {
      for(int i = 0; i < e.length; i++)
         afinar(e[i]);
   }
   public static void main(String[] args) {
      Instrumento[] orquesta = new Instrumento[5];
      int i = 0;
      // Conversión hacia arriba durante la inversión en el array:
      orquesta[i++] = new Viento();
      orquesta[i++] = new Percusion();
      orquesta[i++] = new Cuerda();
      orquesta[i++] = new Metal();
      orquesta[i++] = new Maderaviento();
      afinarTodo(orquesta);
   }
} ///:~
```

Se puede ver que realmente no hay cambios más que en la clase base.

Ayuda crear clases y métodos **abstractos** porque hacen que esa abstracción de la clase sea explícita, e indican, tanto al usuario, como al compilador cómo se tiene que usar.

Constructores y polimorfismo

Como es habitual, los constructores son distintos de otros tipos de métodos. Esto también es cierto cuando se ve involucrado el polimorfismo. Incluso aunque los constructores no sean polimórficos (aunque se puede tener algún tipo de "constructor virtual", como se verá en el Capítulo 12), es importante entender la forma en que trabajan los constructores en jerarquías complejas y con polimorfismo. Esta idea ayudará a evitar posteriores problemas.

Orden de llamadas a constructores

El orden de las llamadas a los constructores se comentó brevemente en el Capítulo 4, y de nuevo en el Capítulo 6, pero esto fue antes de introducir el polimorfismo.

En el constructor de una clase derivada siempre se invoca a un constructor de la clase base, encadenando la jerarquía de herencias de forma que se invoca a un constructor de cada clase base. Esto tiene sentido porque el constructor tiene un trabajo especial: ver que el objeto se ha construido correctamente. Una clase derivada tiene acceso, sólo a sus propios miembros, y no a aquéllos de la clase base (cuyos miembros suelen ser generalmente **privados**). Sólo el constructor de la clase base tiene el conocimiento adecuado y el acceso correcto para inicializar sus propios elementos. Por consiguiente, es esencial que se llegue a invocar a todos los constructores, si no, no se construiría

el objeto entero. Ésta es la razón por la que el compilador realiza una llamada al constructor por cada una de las clases derivadas. Si no se llama explícitamente al constructor de la clase base en el cuerpo del constructor de la clase derivada, llamará al constructor por defecto. Si no hay constructor por defecto, el compilador se quejará. (En el caso en que una clase no tenga constructores, el compilador creará un constructor por defecto automáticamente.)

Echemos un vistazo a un ejemplo que muestra los efectos de la composición, la herencia y el polimorfismo en el orden de construcción:

```
//: c07:Bocadillo.java
// Orden de llamadas a constructores.

class Comida {
  Comida() { System.out.println("Comida()"); }
}

class Pan {
  Pan() { System.out.println("Pan()"); }
}

class Queso {
  Queso() { System.out.println("Queso()"); }
}

class Lechuga {
  Lechuga() { System.out.println("Lechuga()"); }
}

class Almuerzo extends Comida {
  Almuerzo() { System.out.println("Almuerzo()");}
}

class AlmuerzoPortable extends Almuerzo {
  AlmuerzoPortable() {
    System.out.println("AlmuerzoPortable ()");
  }
}

class Bocadillo extends AlmuerzoPortable {
  Pan b = new Pan();
  Queso c = new Queso();
  Lechuga l = new Lechuga();
  Bocadillo() {
    System.out.println("Bocadillo()");
  }
```

```
  public static void main(String[] args) {
    new Bocadillo();
  }
} ///:~
```

Este ejemplo crea una clase compleja a partir de las otras clases, y cada clase tiene un constructor que la anuncia a sí misma. La clase principal es **Bocadillo**, que refleja tres niveles de herencia (cuatro, si se cuenta la herencia implícita de **Object**) y tres objetos miembro. Cuando se crea un objeto **Bocadillo** en el método **main( )**, la salida es:

```
Comida()
Almuerzo()
AlmuerzoPortable()
Pan()
Queso()
Lechuga()
Bocadillo()
```

Esto significa que el orden de las llamadas al constructor para un objeto completo es como sigue:

1. Se invoca al constructor de la clase base. Este paso se repite recursivamente de forma que se construya primero la raíz de la jerarquía, seguida de la siguiente clase derivada, etc. y así hasta que se llega a la última clase derivada.

2. Se llama a los inicializadores de miembros en el orden de declaración.

3. Se llama al cuerpo del constructor de la clase derivada.

El orden de las llamadas a los constructores es importante. Cuando se hereda, se sabe todo lo relativo a la clase base y se puede acceder a cualquier miembro **público** y **protegido** de la clase base. Esto significa que debemos ser capaces de asumir que todos los miembros de la clase base sean válidos cuando se está en la clase derivada. En un método normal, la construcción ya ha tenido lugar, de forma que se han construido todos los miembros de todas las partes del objeto. Dentro del constructor, sin embargo, hay que ser capaz de asumir que se han construido todos los miembros que se usan. La única garantía de esto es que se llame primero al constructor de la clase base. Después, en el constructor de la clase derivada, se inicializarán todos los miembros a los que se puede acceder en la clase base. "Saber que son válidos todos los miembros" dentro del constructor es otra razón para, cuando sea posible, inicializar todos los objetos miembro (es decir, los objetos ubicados en la clase utilizando la composición) al definir la clase (por ejemplo, b, c y l en el ejemplo anterior). Si se sigue esta práctica, se ayudará a asegurar que se han inicializado todos los miembros de la clase base y los objetos miembro del objeto actual. Desgraciadamente, esto no gestiona todos los casos, como se verá en la siguiente sección.

Herencia y **finalize( )**

Cuando se usa composición para crear una clase nueva, no hay que preocuparse nunca de finalizar los objetos miembros de esa clase. Cada miembro es un objeto independiente, y por consiguiente, será eliminado por el recolector de basura de modo independiente. Con la herencia, sin embargo,

hay que superponer el método **finalize( )** de la clase derivada si se tiene alguna limpieza especial a realizar como parte de la recolección de basura. Cuando se superpone el método **finalize( )** en una clase heredada, es importante que recordemos invocar a la versión de la clase base de **finalize( )**, puesto que de otra forma no se finalizará la clase base. El siguiente ejemplo lo prueba:

```
//: c07:Rana.java
// Probando finalize con herencia.

class HacerFinalizacionBase {
  public static boolean indicador = false;
}

class Caracteristica {
  String s;
  Caracteristica(String c) {
    s = c;
    System.out.println(
      "Creando Caracteristica " + s);
  }
  protected void finalize() {
    System.out.println(
      "finalizando Caracteristica " + s);
  }
}

class CriaturaViviente {
  Caracteristica p =
    new Caracteristica("esta vivo");
  CriaturaViviente() {
    System.out.println("CriaturaViviente()");
  }
  protected void finalize() throws Throwable {
    System.out.println(
      "Finalizando CriaturaViviente");
    // ¡Llamar a la versión de la clase base al FINAL!
    if(HacerFinalizacionBase.indicador)
      super.finalize();
  }
}

class Animal extends CriaturaViviente {
  Caracteristica p =
    new Caracteristica("tiene corazon");
  Animal() {
```

```
      System.out.println("Animal()");
   }
   protected void finalize() throws Throwable {
      System.out.println("Finalizando Animal");
      if(HacerFinalizacionBase.indicador)
         super.finalize();
   }
}

class Anfibio extends Animal {
   Caracteristica p =
      new Caracteristica("puede vivir en el agua");
   Anfibio() {
      System.out.println("Anfibio()");
   }
   protected void finalize() throws Throwable {
      System.out.println("Finalizando Anfibio");
      if(HacerFinalizacionBase.indicador)
         super.finalize();
   }
}

public class Rana extends Anfibio {
   Rana() {
      System.out.println("Rana()");
   }
   protected void finalize() throws Throwable {
      System.out.println("Finalizando Rana");
      if(HacerFinalizacionBase.indicador)
         super.finalize();
   }
   public static void main(String[] args) {
      if(args.length != 0 &&
         args[0].equals("finalizar"))
         HacerFinalizacionBase.indicador = true;
      else
         System.out.println("No finalizando las bases");
      new Rana(); // Se convierte en basura automáticamente
      System.out.println("¡Adios!");
      // Forzar la invocación de todas las funciones:
      System.gc();
   }
} ///:~
```

La clase **HacerFinalizacionBase** simplemente guarda un *indicador* que informa a cada clase de la jerarquía si debe llamar a **super.finalize( )**. Este *indicador* se pone a uno con un parámetro de línea de comandos, de forma que se puede ver el comportamiento con y sin finalización de la clase base.

Cada clase de la jerarquía también contiene un objeto miembro de clase **Característica**. Se verá que independientemente de si se llama a los finalizadores de la clase base, siempre se finalizan los objetos miembros **Característica**.

Cada **finalize( )** superpuesto debe tener acceso, al menos a los miembros **protegidos** puesto que el método **finalize( )** de la clase **Object** es **protegido** y el compilador no permitirá reducir el acceso durante la herencia. ("Amistoso" es menos accesible que **protegido.**)

En **Rana.main( )**, se configura el indicador de **HacerFinalizacionBase** y se crea un único objeto **Rana**. Recuerde que el recolector de basura —y en particular la finalización— podrían no darse para algún objeto en particular, por lo que para fortalecer la limpieza, la llamada a **System.gc( )** dispara el recolector de basura, y por consiguiente, la finalización. Sin la finalización de la clase base, la salida es:

```
No finalizando las bases
Creando Caracteristica esta vivo
CriaturaViviente()
Creando Caracteristica tiene corazon
Animal()
Creando Caracteristica puede vivir en el agua
Anfibio()
Rana()
¡Adios!
finalizando Rana
finalizando Caracteristica esta vivo
finalizando Caracteristica tiene corazon
finalizando Caracteristica puede vivir en el agua
```

Se puede ver que, sin duda, no se llama a los finalizadores para las clases base de **Rana** (los miembros objeto *son* finalizados, como se esperaba). Pero si se añade el parámetro "finalizar" en la línea de comandos, se tiene:

```
Creando Caracteristica esta vivo
CriaturaViviente()
Creando Caracteristica tiene corazon
Animal()
Creando Caracteristica puede vivir en el agua
Anfibio()
Rana()
¡Adios!
finalizando Rana
finalizando Anfibio
finalizando CriaturaViviente
Finalizando Caracteristica esta vivo
```

```
finalizando Caracteristica tiene corazon
finalizando Caracteristica puede vivir en el agua
```

Aunque el orden en que finalizan los objetos miembro es el mismo que el de su creación, técnicamente el orden de finalización de los objetos no se especifica. Con las clases base, sin embargo, se tiene control sobre el orden de finalización. El mejor a usar es el que se muestra aquí, que es el inverso al orden de inicialización. Siguiendo la forma que se usa en los destructores de C++, se debería hacer primero la finalización de la clase derivada, y después la finalización de la clase base. Esto es porque la finalización de la clase derivada podría llamar a varios métodos de la clase base que requieran que los componentes de la clase base sigan vivos, por lo que no hay que destruirlos prematuramente.

Comportamiento de métodos polimórficos dentro de constructores

La jerarquía de llamadas a constructores presenta un interesante dilema. ¿Qué ocurre si uno está dentro de un constructor y se invoca a un método de ligadura dinámica del objeto que se está construyendo? Dentro de un método ordinario se puede imaginar lo que ocurrirá —la llamada que conlleva una ligadura dinámica se resuelve en tiempo de ejecución, pues el objeto no puede saber si pertenece a la clase dentro de la que está el método o a alguna clase derivada de ésta. Por consistencia, se podría pensar que esto es lo que debería pasar dentro de los constructores.

Éste no es exactamente el caso. Si se invoca a un método de ligadura dinámica dentro de un constructor, se utiliza la definición superpuesta de ese método. Sin embargo, el *efecto* puede ser bastante inesperado, y puede ocultar errores difíciles de encontrar.

Conceptualmente, el trabajo del constructor es crear el objeto (lo que es casi una proeza). Dentro de cualquier constructor, el objeto entero podría formarse sólo parcialmente —sólo se puede saber que se han inicializado los objetos de clase base, pero no se puede saber qué clases se heredan. Una llamada a un método de ligadura dinámica, sin embargo, se "sale" de la jerarquía de herencias. Llama a un método de una clase derivada. Si se hace esto dentro del constructor, se llama a un método que podría manipular miembros que no han sido aún inicializados —lo que ocasionará problemas.

Se puede ver este problema en el siguiente ejemplo:

```
//: c07:ConstructoresMultiples.java
// Los constructores y el polimorfismo
// no producen lo que cabría esperar.

abstract class Grafica {
  abstract void dibujo();
  Grafica() {
    System.out.println("Grafica() antes de dibujar()");
    dibujar();
    System.out.println("Grafica() despues de dibujar()");
```

```
   }
}

class GraficaCircular extends Grafica {
   int radio = 1;
   GraficaCircular(int r) {
      radio = r;
      System.out.println(
         "GraficaCircular.GraficaCircular(), radio = "
         + radio);
   }
   void dibujar() {
      System.out.println(
         "GraficaCircular.dibujar(), radio = " + radio);
   }
}

public class PoliConstructors {
   public static void main(String[] args) {
      new GraficaCircular(5);
   }
} ///:~
```

En **Gráfica**, el método **dibujar( )** es **abstracto**, pero está diseñado para ser superpuesto. Sin duda, uno se ve forzado a superponerlo en **GraficaCircular**. Pero el constructor **Gráfica** llama a este método, y la llamada acaba en **GraficaCircular.dibujar( )**, que podría parecer ser lo pretendido. Pero, veamos la salida:

```
Grafica() antes de dibujar()
GraficaCircular.dibujar(), radio = 0
Grafica() despues de dibujar()
GraficaCircular.GraficaCircular(), radio = 5
```

Cuando el constructor **Gráfica( )** llama a **dibujar( )**, el valor de **radio** ni siquiera es el valor inicial por defecto 1. Es 0. Esto podría provocar que se dibuje un punto en la pantalla, dejándole a uno atónito, tratando de averiguar por qué el programa no funciona.

El orden de inicialización descrito en la sección previa no es del todo completo, y ahí está la clave para solucionar el misterio. De hecho, el proceso de inicialización es:

1. Se inicializa el espacio de almacenamiento asignado al objeto a ceros binarios antes de que ocurra nada más.

2. Se invoca a los constructores de clase base como se describió previamente. En este momento, se invoca al método **dibujar( )** superpuesto (sí, *antes* de que se invoque al constructor **GraficaCircular**) que descubre un valor de cero para **radio**, debido al punto 1.

3. Se llama a los inicializadores de los miembros en el orden de declaración.
4. Se invoca al cuerpo del constructor de la clase derivada.

Hay algo positivo en todo esto, y es que todo se inicializa al menos a cero (o lo que cero sea para cada tipo de datos en particular) y no se deja simplemente como si fuera basura. Esto incluye referencias a objetos empotradas dentro de clases a través de composición, que se convertirán en **null**. Por tanto, si se olvida inicializar esa referencia, se logrará una referencia en tiempo de ejecución. Todo lo demás se pone a cero, lo que generalmente es un valor revelador al estudiar la salida.

Por otro lado, uno podría acabar horrorizado al ver la salida de su programa. Se ha hecho algo perfectamente lógico, sin quejas por parte del compilador, y sin embargo el comportamiento es misteriosamente erróneo. (C++ produce un comportamiento bastante más racional en esta situación.) Fallos como éste podrían quedar fácilmente enterrados y llevaría mucho tiempo descubrirlos.

Como resultado, una buena guía para los constructores sería, "Haz lo menos posible para dejar el objeto en un buen estado, y en la medida de lo posible, no llames a ningún método". Los únicos métodos seguros a los que se puede llamar dentro de un constructor son aquéllos que sean **constantes** dentro de la clase base. (Esto también se aplica a métodos **privados**, que son automáticamente **constantes**). Éstos no pueden ser superpuestos, y por consiguiente, no pueden producir este tipo de sorpresa.

Diseño con herencia

Una vez que se ha aprendido lo relativo al polimorfismo, puede parecer que todo debería ser heredado, siendo como es el polimorfismo una herramienta tan inteligente. Esto puede cargar los diseños; de hecho, si se elige la herencia en primer lugar cuando se esté usando una clase para construir otra nueva, las cosas pueden volverse innecesariamente complicadas.

Un mejor enfoque es elegir primero la composición, cuando no es obvio qué es lo que debería usarse. La composición no fuerza un diseño en una jerarquía de herencias. Y además, es más flexible dado que es posible elegir dinámicamente un tipo (y por tanto, un comportamiento) al usar la composición, mientras que la herencia requiere conocer un tipo exacto en tiempo de compilación. El ejemplo siguiente lo ilustra:

```
//: c07:Transformar.java
// Cambia dinámicamente el comportamiento de
// un objeto a través de la composición.

abstract class Actor {
  abstract void actuar();
}

class ActorFeliz extends Actor {
  public void actuar() {
    System.out.println("ActorFeliz");
  }
```

```
}

class ActorTriste extends Actor {
  public void actuar() {
    System.out.println("ActorTriste");
  }
}

class Escenario {
  Actor a = new ActorFeliz();
  void cambiar() { a = new ActorTriste(); }
  void ir() { a.actuar(); }
}

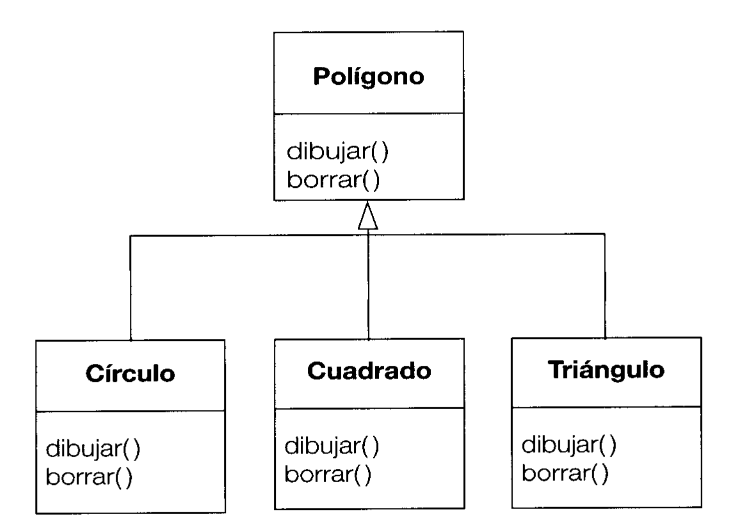
public class Transformar {
  public static void main(String[] args) {
    Escenario s = new Escenario();
    s.ir(); // Imprime "ActorFeliz"
    s.cambiar();
    s.cambiar(); // Imprime "ActorTriste"
  }
} ///:~
```

Un objeto **Escenario** contiene una referencia a un **Actor**, que se inicializa a un objeto **ActorFeliz**. Esto significa que **ir( )** produce un comportamiento particular. Pero dado que se puede reasignar una referencia a un objeto distinto en tiempo de ejecución, en escenario **a** puede sustituirse por una referencia a un objeto **ActorTriste**, con lo que cambia el comportamiento producido por **ir( )**. Por tanto, se gana flexibilidad dinámica en tiempo de ejecución. (A esto también se le llama el *Patrón Estado*. Véase *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*). Por contra, no se puede decidir heredar de forma distinta en tiempo de ejecución; eso debe determinarse completamente en tiempo de compilación.

Una guía general es "Utilice la herencia para expresar diferencias de comportamiento, y campos para expresar variaciones de estado". En el ejemplo de arriba, se usan ambos: se heredan dos clases distintas para expresar la diferencia en el método **actuar( )**, y **Escenario** usa la composición para permitir que varíe su estado. En este caso, ese cambio de estado viene a producir un cambio de comportamiento.

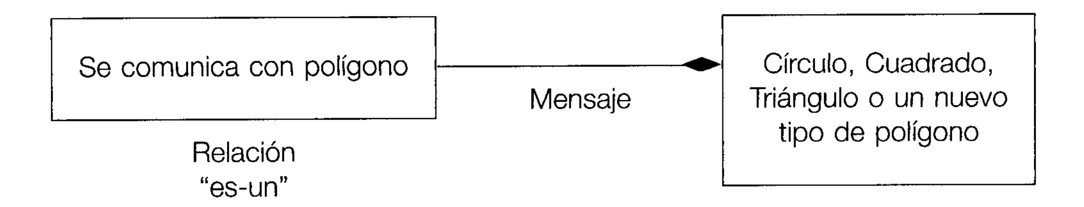
Herencia pura frente a extensión

Cuando se estudia la herencia, podría parecer que la forma más limpia de crear una jerarquía de herencias es seguir el enfoque "puro". Es decir, sólo se pueden superponer en la clase derivada los métodos que se han establecido en la clase base o la **interfaz**, como se muestra en este diagrama:



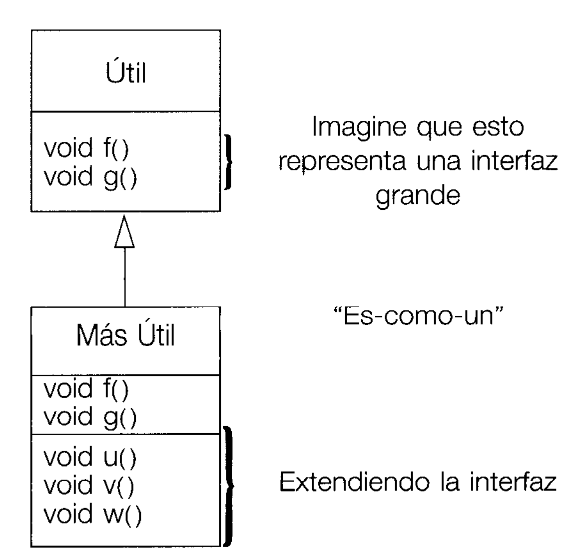
Se podría decir que ésta es una relación "es-un" pura porque el interfaz de la clase establece qué es. La herencia garantiza que cualquier clase derivada tendrá la interfaz de la clase base. Si se sigue el diagrama de arriba, las clases derivadas tampoco tendrán *nada más* que la interfaz de la clase base.

Podría pensarse que esto es una *sustitución pura*, porque los objetos de la clase derivada pueden sustituir perfectamente a la clase base, no siendo necesario conocer en estos casos ninguna información extra de las subclases cuando éstas se usan.

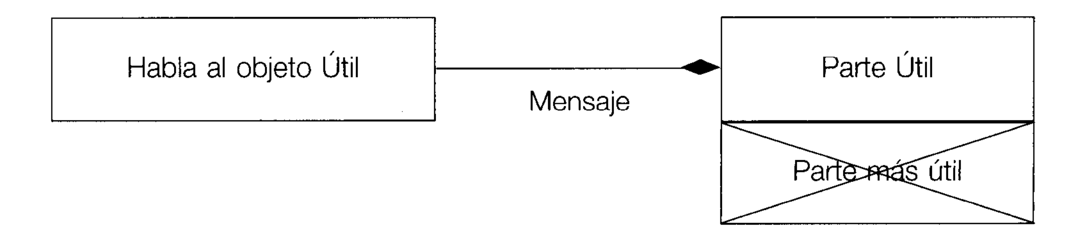


Es decir, la clase base puede recibir cualquier mensaje que se pueda enviar a la clase derivada porque ambas tienen exactamente la misma interfaz. Todo lo que se necesita es hacer una conversión hacia arriba desde la clase derivada y nunca volver a mirar con qué tipo exacto de objeto se está tratando. Todo se maneja mediante el polimorfismo.

Cuando se ve esto así, parece que una relación "es-un" pura es la única manera sensata de hacer las cosas, y cualquier otro diseño indica pensamiento desordenado y es por definición, un problema. Esto también es una trampa. En cuanto se empieza a pensar así, uno llega a descubrir que extender la interfaz (a lo que desafortunadamente parece animar la palabra clave **extends**) es la solución perfecta a un problema particular. Esto podría denominarse relación "es-como-un" porque la clase derivada es *como* la clase base —tiene la misma interfaz fundamental— pero tiene otros aspectos que requieren la implementación de métodos adicionales:



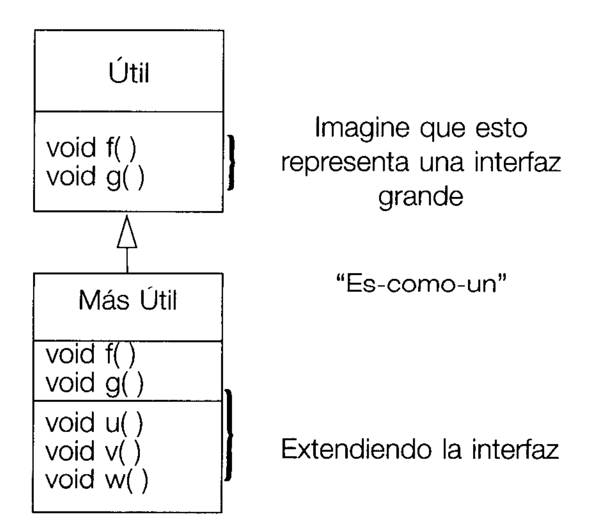
Mientras éste es también un enfoque sensato (dependiendo de la situación) tiene su desventaja. La parte extendida de la interfaz de la clase derivada no está disponible desde la clase base, de forma que una vez que se hace la conversión hacia arriba, no se puede invocar a los nuevos métodos:



Si en este caso no se está haciendo una conversión hacia arriba, no importa, pero a menudo nos meteremos en una situación en la que son necesario redescubrir el tipo exacto del objeto de forma que se pueda acceder a los métodos extendidos de ese tipo. La sección siguiente muestra cómo se ha hecho esto.

Conversión hacia abajo e identificación de tipos en tiempo de ejecución

Dado que a través de una conversión *hacia arriba* se pierde información específica de tipos (al moverse hacia arriba por la jerarquía), tiene sentido hacer una conversión hacia abajo si se quiere recuperar la información de tipos —es decir, moverse de nuevo *hacia abajo* por la jerarquía. Sin embargo, se sabe que una conversión hacia arriba es siempre segura; la clase base no puede tener una interfaz mayor que la de la clase derivada, por consiguiente, se garantiza que todo mensaje que se envíe a través de la interfaz de la clase base sea aceptado. Pero con una conversión hacia abajo, verdaderamente no se sabe que un polígono (por ejemplo) es, de hecho, un círculo. En vez de esto, podría ser un triángulo, un cuadrado o cualquier otro tipo.



Para solucionar este problema, debe haber alguna manera de garantizar que una conversión hacia abajo sea correcta, de forma que no se hará una conversión accidental a un tipo erróneo, para después enviar un mensaje que el objeto no pueda aceptar. Esto sería bastante inseguro.

En algunos lenguajes (como C++) hay que llevar a cabo una operación especial para conseguir una conversión hacia abajo segura en lo que a tipos se refiere, pero en Java ¡se comprueban *todas las conversiones*! Así, aunque parece que se está llevando a cabo una conversión ordinaria entre paréntesis, en tiempo de ejecución se comprueba esta conversión para asegurar que, de hecho, es del tipo que se cree que es. Si no lo es, se obtiene una **ClassCastException**. A esta comprobación de tipos en tiempo de ejecución se le denomina *identificación de tipos en tiempo de ejecución*[2]. El ejemplo siguiente demuestra el comportamiento de esta identificación de tipos:

```
//: c07:ITTE.java
// Conversión hacia abajo e Identificación de tipos
// en Tiempo de ejecución (ITTE)
import java.util.*;

class Util {
  public void f() {}
  public void g() {}
}

class MasUtil extends Util {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
```

[2] N. de T.: RTTI: Run-Time Type Identification.

```
}

public class ITTE {
   public static void main(String[] args) {
      Util[] x = {
         new Util(),
         new MasUtil()
      };
      x[0].f();
      x[1].g();
      // Tiempo de compilación: método no encontrado en útil:
      //! x[1].u();
      ((MasUtil)x[1]).u(); // Conversión hacia abajo/ITTE
      ((MasUtil)x[0]).u(); // Se lanza una Excepción
   }
} ///:~
```

Como en el diagrama, **MasUtil** extiende la interfaz de **Util**. Pero dado que es heredada, también puede tener una conversión hacia arriba hasta **Util**. Se puede ver cómo ocurre esto en la inicialización del array **x** en **main( )**. Dado que ambos objetos del array son de clase **Util**, se pueden enviar los métodos **f( )** y **g( )** a ambos, y si se intenta llamar a **u( )** (que sólo existe en **MasUtil**) se obtendrá un mensaje de error de tiempo de compilación.

Si se desea acceder a la interfaz extendida de un objeto **MasUtil**, se puede intentar hacer una conversión hacia abajo. Si es del tipo correcto, tendrá éxito. En caso contrario, se obtendrá una **ClassCastException**. No es necesario escribir ningún código extra para esta excepción, dado que indica un error del programador que podría ocurrir en cualquier lugar del programa.

Hay más que una simple conversión en la identificación de tipos en tiempo de ejecución. Por ejemplo, hay una forma de ver con qué tipo se está tratando *antes* de intentar hacer una conversión hacia abajo. Todo el Capítulo 12 está dedicado al estudio de distintos aspectos de la identificación de tipos en tiempo de ejecución de Java.

Resumen

Polimorfismo quiere decir "formas diferentes". En la programación orientada a objetos se tiene un mismo rostro (la interfaz común de la clase base) y distintas formas de usar ese rostro: las diferentes versiones de los métodos de la ligadura dinámica.

Hemos visto en este capítulo que es imposible entender, o incluso crear, un ejemplo de polimorfismo sin utilizar abstracción de datos y herencia. El polimorfismo es una faceta que no se puede ver aislada (como sí se podría hacer una sentencia **switch**, por ejemplo), pero sin embargo, funciona sólo dentro de un contexto, como parte de la "gran figura" que conforman las relaciones de clases. Las personas suelen confundirse con otras características de Java no orientadas a objetos, como la

sobrecarga de métodos, que en ocasiones se presenta como orientada a objetos. No se dejen engañar: si no hay ligadura tardía, no hay polimorfismo.

Para usar polimorfismo —y por consiguiente, técnicas de orientación a objetos— de manera efectiva en los programas, hay que ampliar la visión que se tiene de la programación para que incluya no sólo a los miembros y mensajes de una clase individual, sino también a aquellos elementos comunes a distintas clases, y sus inter-relaciones. Aunque esto requiere de un esfuerzo significativo, merece la pena, pues los resultados son un desarrollo de programas más rápido, una mejor organización del código, programas ampliables y un mantenimiento más sencillo del código.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com.*

1. Añadir un nuevo método a la clase base de **Poligonos.java** que imprima un mensaje, pero no superponerlo en la clase derivada. Explicar lo que ocurre. Ahora superponerlo en una de las clases derivadas pero no en las otras. Ver qué ocurre. Finalmente, superponerlo en todas las clases derivadas.

2. Añadir un nuevo tipo de **Polígono** a **Poligonos.java** y verificar en el método **main( )** que el polimorfismo funciona para el nuevo tipo como lo hace para los viejos.

3. Cambiar **Musica3.java** de forma que el método **que( )** se convierta en el método **toString( )** del objeto raíz **Object**. Intentar imprimir los objetos **Instrumento** haciendo uso de **System.out.println( )** (sin hacer uso de ningún tipo de conversión).

4. Añadir un nuevo tipo de **Instrumento** a **Musica3.java** y verificar que el polimorfismo funciona para el nuevo tipo.

5. Modificar **Musica3.java** de forma que cree objetos **Instrumento** al azar de la misma manera que lo hace **Poligonos.java**.

6. Crear una jerarquía de herencia de **Roedor**: **Ratón, Jerbo**, **Hamster**, etc. En la clase base, proporcionar métodos comunes a todas las clases de tipo **Roedor**, y superponerlos en las clases derivadas para que lleven a cabo distintos comportamientos en función del tipo de **Roedor** específico. Crear un array de objetos de tipo **Roedor**, rellenarlo con tipos de **Roedor** diferentes, e invocar a los métodos de la clase base para ver qué pasa.

7. Modificar el Ejercicio 6 de forma que **Roedor** sea una clase **abstracta**. Convertir en **abstractos** todos los métodos de **Roedor** que sea posible.

8. Crear una clase **abstracta** sin incluir ningún método **abstracto**, y verificar que no se pueden crear instancias de esa clase.

9. Añadir una clase **Escabeche** a **Bocadillo.java**.

10. Modificar el Ejercicio 6, de forma que demuestre el orden de inicialización de las clases base y las clases derivadas. Ahora, añadir objetos miembros, tanto a la clase base, como a las derivadas, y mostrar el orden en que se da la inicialización durante su construcción.

11. Crear una jerarquía de herencia de 3 niveles. Cada clase de la jerarquía debería tener un método **finalize( )**, y debería llamar adecuadamente a la versión de **finalize( )** de la clase base. Demostrar que la jerarquía funciona adecuadamente.

12. Crear una clase base con dos métodos. En el primer método, llamar al segundo método. Heredar una clase y superponer el segundo método. Crear un objeto de la clase derivada, hacer una conversión hacia arriba de la misma al tipo base, e invocar al primer método. Explicar lo que ocurre.

13. Crear una clase base con un método **abstracto escribir( )** superpuesta en una clase derivada. La versión superpuesta del método imprime el valor de una variable **entera** definida en la clase derivada. En el momento de la definición de esta variable, darle un valor distinto de cero. En el constructor de la clase base, invocar a este método. En el método **main( )** crear un objeto del tipo derivado, y después llamar a su método **escribir( )**. Explicar los resultados.

14. Siguiendo el ejemplo de **Transformar.java**, crear una clase **Estrella** que contenga una referencia a **EstadosAlerta** que pueda indicar tres estados diferentes. Incluir métodos para cambiar los estados.

15. Crear una clase **abstracta** sin métodos. Derivar una clase y añadir un método. Crear un método **estático** que toma una referencia a la clase base, haga una conversión hacia abajo y llame al método. Demostrar que funciona utilizando el método **main( )**. Ahora poner la declaración **abstracta** del método en la clase base, eliminando por consiguiente la necesidad de la conversión hacia abajo.

8: Interfaces y clases internas

Los interfaces y las clases internas proporcionan formas más sofisticadas de organizar y controlar los objetos de un sistema.

C++, por ejemplo, no contiene estos mecanismos, aunque un programador inteligente podría simularlos. El hecho de que existan en Java indica que se consideraban lo suficientemente importantes como para proporcionarles soporte directo en forma de palabras clave del lenguaje.

En el Capítulo 7, se aprendió lo relativo a la palabra clave **abstract**, que permite crear uno o más métodos sin definición dentro de una clase —se proporciona parte de la interfaz sin proporcionar la implementación correspondiente, que será creada por sus descendientes. La palabra clave **interface** produce una clase completamente abstracta, que no tiene ningún tipo de implementación. Se aprenderá que una **interfaz** es más que una clase abstracta llevada al extremo, pues permite llevar a cabo una variación de la "herencia múltiple" de C++, creando una clase sobre la que se puede hacer conversión hacia arriba a más de un tipo base.

Al principio, las clases internas parecen como un simple mecanismo de ocultación de código: se ubican clases dentro de otras clases. Se aprenderá, sin embargo, que una clase interna hace más que esto —conoce y puede comunicarse con las clases que le rodean— y el tipo de código que se puede escribir con clases internas es más elegante y claro, aunque para la mayoría de personas constituye un concepto nuevo. Lleva algún tiempo habituarse al diseño haciendo uso de clases internas.

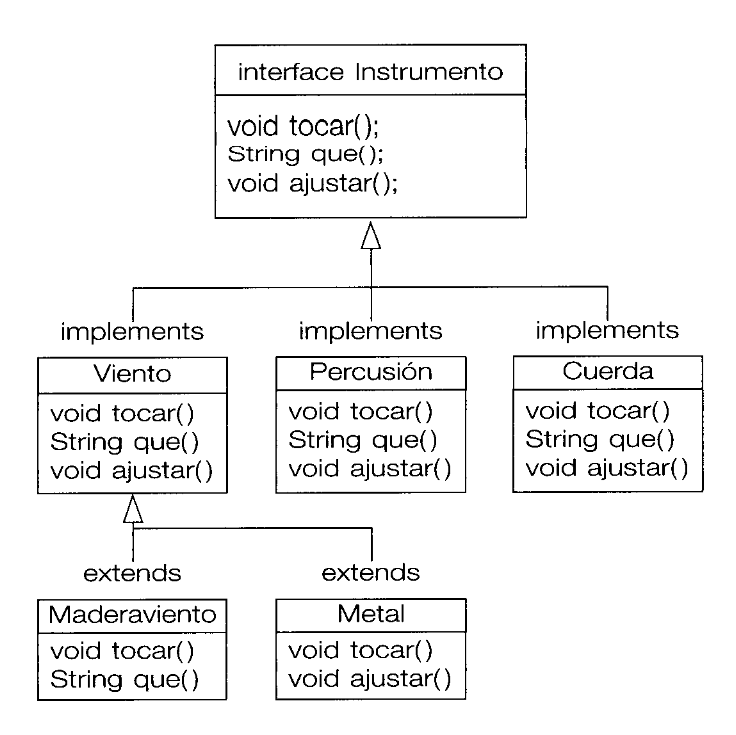
Interfaces

La palabra clave **interfaz** lleva el concepto de **abstracción** un paso más allá. Se podría pensar que es una clase **abstracta** "pura". Permite al creador establecer la forma de una clase: nombres de métodos, listas de parámetros, y tipos de retorno, pero no cuerpos de métodos. Una **interfaz** también puede contener campos, pero éstos son implícitamente **estáticos** y **constantes**. Una **interfaz** proporciona sólo la forma, pero no la implementación.

Una **interfaz** dice: "Ésta es la apariencia que tendrán todas las clases que implementen esta **interfaz**". Por consiguiente, cualquier código que use una **interfaz** particular sabe qué métodos deberían ser invocados por esa **interfaz**, y eso es todo. Por tanto se usa la **interfaz** para establecer un "protocolo" entre clases. (Algunos lenguajes de programación orientada a objetos tienen la palabra clave *protocolo* para hacer lo mismo.)

Para crear una **interfaz**, se usa la palabra clave **interface** en vez de la palabra clave **class**. Al igual que una clase, se le puede anteponer la palabra **public** a **interface** (pero sólo si esa **interfaz** se definió en un archivo con el mismo nombre), o dejar que se le dé el status de "amistoso" de forma que sólo se podrá usar dentro del mismo paquete.

Para hacer una clase que se ajuste a una **interfaz** particular (o a un grupo de **interfaces**), se usa la palabra clave **implements**. Se está diciendo "La **interfaz** contiene la apariencia, pero ahora voy a decir cómo *funciona*". Por lo demás, es como la herencia. El diagrama del ejemplo de los instrumentos lo muestra:



Una vez implementada una **interfaz**, esa implementación se convierte en una clase ordinaria que puede extenderse de forma normal.

Se puede elegir manifestar explícitamente las declaraciones de métodos de una **interfaz** como **pública**. Pero son **públicas** incluso si no se dice. Por tanto, cuando se **implementa** una **interfaz**, deben definirse como **públicos** los métodos de la **interfaz**. De otra forma, se pondrían por defecto a "amistoso", y se estaría reduciendo la accesibilidad a un método durante la herencia, lo que no permite el compilador de Java.

Se puede ver esto en la versión modificada del ejemplo **Instrumento**. Fíjese que todo método de la **interfaz** es estrictamente una declaración, que es lo único que permite el compilador. Además, ninguno de los métodos de **Instrumento** se declara como **público**, pero son **público** automáticamente:

```
//: c08:musica5:Musica5.java
// Interfaces.
import java.util.*;

interface Instrumento {
```

```
    // Tiempo de compilación constante:
    int i = 5; // estático y constante
    // No puede tener definiciones de métodos:
    void tocar(); // Automáticamente public
    String que();
    void ajustar();
}

class Viento implements Instrumento {
    public void tocar() {
        System.out.println("Viento.tocar()");
    }
    public String que() { return "Viento"; }
    public void ajustar() {}
}

class Percusion implements Instrumento {
    public void tocar() {
        System.out.println("Percusion.tocar()");
    }
    public String que() { return "Percusion"; }
    public void ajustar() {}
}

class Cuerda implements Instrumento {
    public void tocar() {
        System.out.println("Cuerda.tocar()");
    }
    public Cuerda que() { return "Cuerda"; }
    public void ajustar() {}
}

class Metal extends Viento {
    public void tocar() {
        System.out.println("Metal.tocar()");
    }
    public void ajustar() {
        System.out.println("Metal.ajustar()");
    }
}

class Maderaviento extends Viento {
    public void tocar() {
        System.out.println("Maderaviento.tocar()");
    }
```

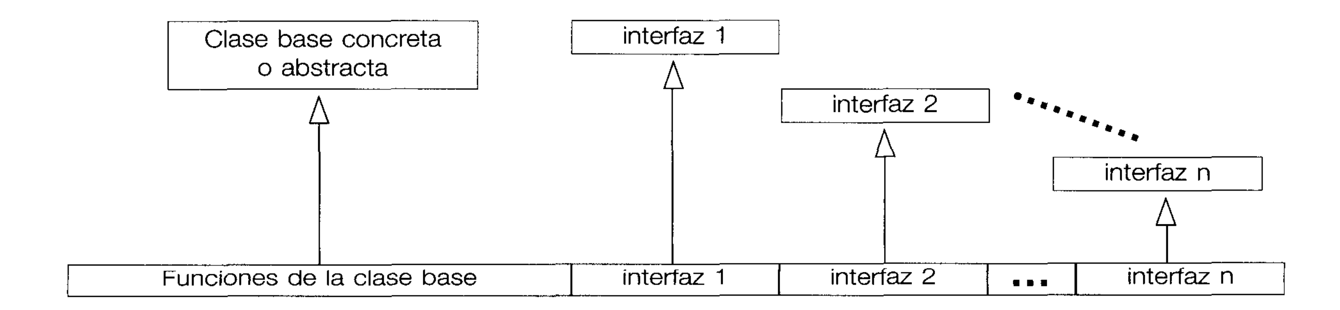
```
   public String que() { return "Maderaviento"; }
}

public class Musica5 {
   // No le importa el tipo por lo que los nuevos
   // tipos que se aniadan al sistema seguirán funcionando bien:
   static void afinar(Instrumento i) {
      // ...
      i.tocar();
   }
   static void afinarTodo(Instrumento[] e) {
      for(int i = 0; i < e.length; i++)
         afinar(e[i]);
   }
   public static void main(String[] args) {
      Instrumento[] orquesta = new Instrumento[5];
      int i = 0;
     // Haciendo conversión hacia arriba durante la inserción en el array:
      orquesta [i++] = new Viento();
      orquesta [i++] = new Percusion();
      orquesta [i++] = new Cuerda();
      orquesta [i++] = new Metal();
      orquesta [i++] = new Maderaviento();
      afinarTodo(orquesta);
   }
} ///:~
```

El resto del código funciona igual. No importa si se está haciendo un conversión hacia arriba a una clase "regular" llamada **Instrumento**, una clase **abstracta** llamada **Instrumento**, o a una **interfaz** denominada **Instrumento**. El comportamiento es el mismo. De hecho, se puede ver en el método **afinar( )** que no hay ninguna prueba de que **Instrumento** sea una clase "regular", una clase **abstracta** o una **interfaz**. Ésta es la intención: cada enfoque da al programador un control distinto sobre la forma de crear y utilizar los objetos.

"Herencia múltiple" en Java

La **interfaz** no es sólo una forma "más pura" de clase **abstracta**. Tiene un propósito mayor. Dado que una **interfaz** no tiene implementación alguna —es decir, no hay espacio de almacenamiento asociado con una **interfaz**— no hay nada que evite que se combinen varias **interfaces**. Esto es muy valioso, pues hay veces en las que es necesario decir "una **x** es una **a** *y* una **b** *y* una **c**". En C++, a este acto de combinar múltiples interfaces de clases se le denomina *herencia múltiple*, y porta un equipaje bastante pegajoso porque puede que cada clase tenga una implementación. En Java, se puede hacer lo mismo, pero sólo una de las clases puede tener una implementación, por lo que los problemas de C++ no ocurren en Java al combinar múltiples interfaces:



En una clase derivada, no hay obligación de tener una clase base que puede ser **abstracta** o "concreta" (aquélla sin métodos **abstractos**). Si se *hereda* desde una no-**interfaz**, se puede heredar sólo de una. Todo el resto de elementos base deben ser **interfaces**. Se colocan todos los nombres de interfaz después de la palabra clave **implements**, separados por comas. Se pueden tener tantas **interfaces** como se desee —cada uno se convierte en un tipo independiente al que se puede hacer conversión hacia arriba. El ejemplo siguiente muestra una clase concreta combinada con varias **interfaces** para producir una nueva clase:

```
//: c08:Aventura.java
// Múltiples interfaces.
import java.util.*;

interface PuedeLuchar {
  void luchar();
}

interface PuedeNadar {
  void nadar();
}

interface PuedeVolar {
  void volar();
}

class PersonajeDeAccion {
  public void luchar() {}
}

class Heroe extends PersonajeDeAccion
    implements PuedeLuchar, PuedeNadar, PuedeVolar {
  public void nadar() {}
  public void volar() {}
}

public class Aventura {
```

```
  static void t(PuedeLuchar x) { x.luchar(); }
  static void u(PuedeNadar x) { x.nadar(); }
  static void v(PuedeVolar x) { x.volar(); }
  static void w(PersonajeDeAccion x) { x.luchar(); }
  public static void main(String[] args) {
    Heroe h = new Heroe();
    t(h); // Tratarlo como un PuedeLuchar
    u(h); // Tratarlo como un PuedeNadar
    v(h); // Tratarlo como un PuedeVolar
    w(h); // Tratarlo como un PersonajeDeAccion
  }
} ///:~
```

Se puede ver que **Héroe** combina la clase concreta **PersonajeDeAccion** con las interfaces **PuedeLuchar**, **PuedeNadar** y **PuedeVolar**. Cuando se combina una clase concreta con interfaces de esta manera, hay que poner primero la clase concreta, y después las interfaces. (Sino, el compilador dará error.)

Fíjese que la sintaxis del método **luchar()** es la misma en la **interfaz PuedeLuchar** y en la clase **PersonajeDeAccion**, y que *no* hay ninguna definición para **luchar()** en **Héroe**. La regla de una **interfaz** es que se puede heredar de ella (como se verá en breve) pero se obtiene otra **interfaz**. Si se desea crear un objeto del nuevo tipo, éste debe ser una clase a la que se proporcionen todas sus definiciones. Incluso aunque **Héroe** no proporciona explícitamente una definición para **luchar( )**, ésta viene junto con **PersonajeDeAccion**, por lo que ésta se proporciona automáticamente y es posible crear objetos de **Heroe**.

En la clase **Aventura**, se puede ver que hay cuatro métodos que toman como parámetros las distintas interfaces y la clase concreta. Cuando se crea un objeto **Héroe**, se le puede pasar a cualquier de estos métodos, lo que significa que se le está haciendo una conversión hacia arriba a cada **interfaz**. Debido a la forma de diseñar las interfaces en Java, esto funciona sin problemas ni esfuerzos por parte del programador.

Recuérdese que la razón principal de las interfaces se muestra en el ejemplo de arriba: poder hacer una conversión hacia arriba a más de un tipo base. Sin embargo, una segunda razón para el uso de interfaces es la misma que se da al usar una clase **abstracta**: evitar que el programador cliente haga objetos de esta clase y hacer que ésta no sea más que una interfaz. Esto provoca una pregunta: ¿debería usarse una **interfaz** o una clase **abstracta**? Una **interfaz** proporciona los beneficios de una clase **abstracta** *y* los beneficios de una **interfaz**, por lo que si es posible crear la clase base sin definiciones de métodos o variables miembro, siempre se debería preferir las **interface**s a las clases **abstractas**. De hecho, si se sabe que algo va a ser una clase base, la primera opción debería ser convertirla en **interfaz**, y sólo si uno se ve forzado a tener definiciones de métodos o variables miembros habrá que cambiar a una clase **abstracta**, o si fuera necesario una clase concreta.

Colisiones de nombre al combinar interfaces

Se puede encontrar una pequeña pega al implementar múltiples interfaces. En el ejemplo de arriba, tanto **PuedeLuchar** como **PersonajeDeAccion** tienen un método **void luchar()** idéntico. Esto no

es un problema al ser el método idéntico en ambos casos pero, ¿qué ocurre si no es así? He aquí un ejemplo:

```
//: c08:ColisionInterfaces.java
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; } // sobrecargado
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // sobrecargado
}

class C4 extends C implements I3 {
  // Idéntica, sin problemas:
  public int f() { return 1; }
}

// Los métodos sólo difieren en el tipo de retorno:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

La dificultad se da porque se mezclan la sobrecarga, la implementación y la superposición, y las funciones sobrecargadas no pueden diferir sólo en el valor de retorno. Cuando se quita el comentario de las dos últimas líneas, los mensajes de error dicen:

```
ColisionInterfaces.java:23: f() in C cannot
implement f() in I1; attempting to use
incompatible return type
found    : int
required: void
ColisionInterfaces.java:24: interfaces I3 and I1 are
incompatible; both define f
(), but with different return type
```

Utilizando los mismos nombres de método en interfaces diferentes que se pretende combinar, suele causar también confusión en la legibilidad del código. Hay que tratar de evitarlo.

Extender una interfaz con herencia

Se pueden añadir nuevas declaraciones de método a una **interfaz** haciendo uso de la herencia, y también se pueden combinar varias **interfaces** en una nueva **interfaz** gracias a la herencia. En ambos casos se consigue una nueva **interfaz**, como se ve en el ejemplo siguiente:

```
//: c08:EspectaculoDeMiedo.java
// Extendiendo una interfaz con herencia.

interface Monstruo {
  void amenaza();
}

interface MonstruoPeligroso extends Monstruo {
  void destruir();
}

interface Letal {
  void matar();
}

class Dragon implements MonstruoPeligroso {
  public void amenaza() {}
  public void destruir() {}
}

interface Vampiro
    extends MonstruoPeligroso, Letal {
  void beberSangre();
}

class EspectaculoDeMiedo {
  static void u(Monstruo b) { b.amenaza(); }
  static void v(MonstruoPeligroso d) {
    d.amenaza();
    d.destuir();
  }
  public static void main(String[] args) {
    Dragon if2 = new Dragon();
    u(if2);
    v(if2);
  }
} ///:~
```

MonstruoPeligroso es una simple extension de **Monstruo** que produce una nueva **interfaz**. Éste se implementa en **Dragon**.

La sintaxis utilizada en **Vampiro** *sólo* funciona cuando se heredan interfaces. Normalmente, se puede usar **herencia** sólo con una única clase, pero dado que una **interfaz** puede estar hecha de otras múltiples interfaces, **extends** puede referirse a múltiples interfaces a base al construir una nueva **interfaz**. Como se puede ver, los nombres de **interfaz** simplemente se separan con comas.

Constantes de agrupamiento

Dado que cualquier campo que se ponga en una interfaz se convierte automáticamente en **estático** y **constante**, la **interface** es una herramienta conveniente para la creación de grupos de valores constantes, en gran medida al igual que se haría con un **enumerado** en C o C++. Por ejemplo:

```
//: c08:Meses.java
// Utilizando interfaces para crear grupos de constantes.
package c08;

public interface Meses {
  int
    ENERO = 1, FEBRERO = 2, MARZO = 3,
    ABRIL = 4, MAYO = 5, JUNIO = 6, JULIO = 7,
    AGOSTO = 8, SEPTIEMBRE = 9, OCTUBRE = 10,
    NOVIEMBRE = 11, DICIEMBRE = 12;
} ///:~
```

Fíjese que el estilo de Java de usar en todo letras mayúsculas (con guiones bajos para separar múltiples palabras dentro de un único identificador) para datos e**státicos** y **constantes** que tienen inicializadores constantes.

Los campos de una **interfaz** son automáticamente **públicos**, por lo que no es necesario especificarlo.

Ahora se pueden usar constantes de fuera del paquete, importándolas de **c08.*** o **c08.Meses** justo como se haría con cualquier otro paquete, y hacer referencia a los valores de expresiones como **Mes.ENERO**. Por supuesto, lo que se consigue es simplemente un **entero**, por lo que no existe la seguridad de tipos extra que tiene el **enumerado** de C++, pero esta técnica (comúnmente usada) es verdaderamente una mejora sobre la codificación ardua de números en un programa. (A este enfoque se le suele denominar cómo hacer uso de "números mágicos" y produce código muy difícil de mantener.)

Si se desea seguridad extra con los tipos, se puede construir una clase como[1]:

```
//: c08:Mes2.java
// Un sistema de enumeracion mas robusto.
package c08;

public final class Mes2 {
  private String nombre;
```

[1] Este enfoque se basa en un e-mail que me envió Rich Hoffarth.

```
  private Mes2(String nm) { nombre = nm; }
  public String toString() { return nombre; }
  public final static Mes2
    ENE = new Mes2("Enero"),
    FEB = new Mes2("Febrero"),
    MAR = new Mes2("Marzo"),
    ABR = new Mes2("Abril"),
    MAY = new Mes2("Mayo"),
    JUN = new Mes2("Junio"),
    JUL = new Mes2("Julio"),
    AGO = new Mes2("Agosto"),
    SEP = new Mes2("Septiembre"),
    OCT = new Mes2("Octubre"),
    NOV = new Mes2("Noviembre"),
    DIC = new Mes2("Diciembre");
  public final static Mes2[] mes =  {
    ENE, ENE, FEB, MAR, ABR, MAY, JUN,
    JUL, AGO, SEP, OCT, NOV, DIC
  };
  public static void main(String[] args) {
    Mes2 m = Mes2.ENE;
    System.out.println(m);
    m = Mes2.mes[12];
    System.out.println(m);
    System.out.println(m == Mes2.DIC);
    System.out.println(m.equals(Mes2.DIC));
  }
} ///:~
```

La clase se llama **Mes2**, dado que ya hay una clase **Mes** (Month) en la biblioteca estándar Java. Es una clase **constante** con un constructor **privado** por lo que nadie puede heredar de la misma o hacer instancias de ella. Las únicas instancias son las **constante estáticas** creadas en la propia clase: **ENE**, **FEB**, **MAR**, etc. Estos objetos también pueden usarse en el array **mes**, que permite elegir los números por número en vez de por nombre. (Fíjese que el **ENE** extra del array proporciona un desplazamiento de uno, de forma que diciembre sea el mes número 12.) En el método **main( )** se puede ver la seguridad de tipos: **m** es un objeto **Mes2**, por lo que puede ser asignado sólo a **Mes2**. El ejemplo anterior **Meses.java** sólo proporcionaba valores **enteros**, por lo que una variable **entera** implementada con el fin de representar un mes, podría recibir cualquier valor entero, lo que no sería muy seguro.

Este enfoque también permite usar == o **equals( )** indistintamente, como se muestra al final del método **main( )**.

Inicializando atributos en interfaces

Los atributos definidos en las interfaces son automáticamente e**státicos** y **constantes**. Éstos no pueden ser "constantes blancas", pero pueden inicializarse con expresiones no constantes. Por ejemplo:

```
//: c08:ValoresAleatorios.java
// Inicializando atributos de interfaz con
// inicializadores no constantes.
import java.util.*;

public interface ValoresAleatorios {
  int rint = (int)(Math.random() * 10);
  long rlong = (long)(Math.random() * 10);
  float rfloat = (float)(Math.random() * 10);
  double rdouble = Math.random() * 10;
} ///:~
```

Dado que los campos son **estáticos**, se inicializan cuando se carga la clase por primera vez, lo que ocurre cuando se accede a cualquiera de los atributos por primera vez. He aquí una simple prueba:

```
//: c08:PruebaValoresAleatorios.java
public class PruebaValoresAleatorios {
  public static void main(String[] args) {
    System.out.println(PruebaValoresAleatorios.rint);
    System.out.println(PruebaValoresAleatorios.rlong);
    System.out.println(PruebaValoresAleatorios.rfloat);
    System.out.println(PruebaValoresAleatorios.rdouble);
  }
} ///:~
```

Los atributos, por supuesto, no son parte de la interfaz, pero se almacenan, sin embargo, en el área de almacenamiento **estático** de esa interfaz.

Interfaces anidados

Se pueden anidar interfaces dentro de clases y dentro de otras interfaces. Esto revela un número de aspectos muy interesantes[2]:

```
//: c08:InterfacesAnidadas.java
class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
```

[2] Gracias a Martin Danner por preguntar esto durante un seminario.

```
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void recibirD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
    void f();
  }
  // "public" es redundante:
  public interface H {
    void f();
  }
  void g();
  // No puede ser privado dentro de una interfaz
  //! private interface I {}
}

public class InterfacesAnidados {
  public class BImp implements A.B {
    public void f() {}
  }
  class CImp implements A.C {
```

```
    public void f() {}
  }
  // No se puede implementar una interfaz anidada excepto que esté
  // dentro de la definición de una clase:
  //! class DImp implements A.D {
  //!   public void f() {}
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
    class EG implements E.G {
      public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // No se puede acceder a A.D:
    //! A.D ad = a.obtenerD();
    // No devuelve nada más que A.D:
    //! A.DImp2 di2 = a.obtenerD();
    // No se puede acceder a un miembro de la interfaz:
    //! a.obtenerD().f();
    // Sólo otro A puede hacer algo con obtenerD():
    A a2 = new A();
    a2.recibirD(a.obtenerD());
  }
} ///:~
```

La sintaxis para anidar una interfaz dentro de una clase es razonadamente obvia, y al igual que con las interfaces no anidadas, éstas pueden tener visibilidad **pública** o "amistosa". También se puede ver que ambas interfaces anidadas **pública** y "amistosa" pueden implementarse como clases anidadas **pública**, "amistosa" y **privada**.

Como novedad, las interfaces también pueden ser **privadas** como se ve en **A.D** (se usa la misma sintaxis de calificaciones que en las clases anidadas).

¿Qué tiene de bueno una interfaz **pública** anidada? Se podría adivinar que sólo puede implementarse como una clase **privada** anidada como en **DImp**, pero **A.DImp2** muestra que también puede implementarse como una clase **pública**. Sin embargo, **A.DImp2** sólo puede ser usada como ella misma. No se permite mencionar el hecho de que implementa la interfaz **privada**, por lo que im-

plementar una interfaz **privada** es una manera de forzar la definición de métodos de esa interfaz sin añadir ninguna información de tipos (es decir, sin permitir conversiones hacia arriba).

El método **obtenerD( )** produce un dilema aún mayor en lo relativo a la interfaz **privada**: es un método **público** que devuelve una referencia a una interface **privada**. ¿Qué se puede hacer con el valor de retorno de este método? En el método **main( )**, se pueden ver varios intentos de usar el valor de retorno, pero todos en balde. Lo único que funciona es pasar el valor de retorno a un objeto que tenga permiso para usarlo —en este caso, otro **A**, a través del método **recibir( )**.

La interfaz E muestra que es posible anidar interfaces una dentro de la otra. Sin embargo, las reglas sobre las interfaces —en particular, que todos los elementos de una interfaz deban ser públicos— se vuelven en este caso muy estrictas, de forma que una interfaz anidada dentro de otra se convierte en **pública** automáticamente y no puede declararse como **privada**.

InterfacesAnidadas muestra las distintas maneras de implementar interfaces anidadas. En particular, fíjese que al implementar una interfaz, no es obligatorio implementar las interfaces que tenga anidadas. Tampoco las interfaces **privadas** pueden implementarse fuera de las clases en que se han definido.

Inicialmente, estas características pueden parecer añadidos para conseguir consistencia sintáctica, pero generalmente encontramos que una vez que se conocen, se descubren a menudo sitios en los que son útiles.

Clases internas

Es posible colocar una definición de clase dentro de otra definición de clase. A la primera se le denomina clase *interna*. Este tipo de clases son una característica valiosa, pues permite agrupar clases que lógicamente están relacionadas, además de controlar la visibilidad de una con la otra. Sin embargo, es importante entender que las clases internas son fundamentalmente distintas de la composición.

A menudo, al aprender clases internas, no se ve su necesidad inmediatamente. Al final de esta sección, una vez que se hayan descrito toda la sintaxis y semántica de las clases internas, se verán ejemplos que deberían aclarar los beneficios de estas clases.

Se crea una clase interna como uno esperaría —ubicando su definición dentro de una clase envolvente:

```
//: c08:Paquete1.java
// Creando clases internas.

public class Paquete1 {
  class Contenidos {
    private int i = 11;
    public int valor() { return i; }
  }
  class Destino {
    private String etiqueta;
    Destino(String aDonde) {
      etiqueta = aDonde;
    }
```

```
      String leerEtiqueta() { return etiqueta; }
    }
    // Usar clases internas es igual que usar
    // otras clases, dentro de Paquete1:
    public void enviar(String dest) {
      Contenidos c = new Contenidos();
      Destino d = new Destino(dest);
      System.out.println(d.leerEtiqueta());
    }
    public static void main(String[] args) {
      Paquete1 p = new Paquete1();
      p.enviar("Tanzania");
    }
} ///:~
```

Las clases internas, cuando se usan dentro de **enviar()** tienen la misma apariencia que muchas otras clases. Aquí, la única diferencia práctica es que los nombres se anidan dentro de **Paquete1**. Se verá en breve que ésta no es la única diferencia.

Generalmente, la clase externa tendrá un método que devuelva una referencia a una clase interna, como ésta:

```
//: c08:Paquete2.java
// Devolviendo una referencia a una clase interna.

public class Paquete2 {
    class Contenidos {
      private int i = 11;
      public int valor() { return i; }
    }
    class Destino {
      private String etiqueta;
      Destino(String aDonde) {
        etiqueta = aDonde;
      }
      String leerEtiqueta() { return etiqueta; }
    }
    public Destino para(String s) {
      return new Destino(s);
    }
    public Contenidos cont() {
      return new Contenidos();
    }
    public void enviar(String dest) {
      Contenidos c = cont();
      Destino d = para(dest);
      System.out.println(d.leerEtiqueta());
```

```
    }
    public static void main(String[] args) {
        Paquete2 p = new Paquete2();
        p.enviar("Tanzania");
        Paquete2 q = new Paquete2();
        // Definir referencias a clases internas:
        Paquete2.Contenidos c = q.cont();
        Paquete2.Destino d = q.para("Borneo");
    }
} ///:~
```

Si se desea hacer un objeto de la clase interna en cualquier sitio que no sea un método no **estático** de la clase externa, hay que especificar el tipo de ese objeto como *NombreClaseExterna.NombreClaseInterna*, como se ha visto en el método **main( )**.

Clases internas y conversiones hacia arriba

Hasta ahora, las clases no parecen excesivamente espectaculares. Después de todo, si uno pretende ocultar, Java ya tiene un buen mecanismo de ocultamiento —simplemente es necesario dejar que la clase sea "amistosa" (visible sólo dentro de un paquete) en vez de crearla como clase interna.

Sin embargo, las clases internas tienen su verdadera razón de ser al comenzar a hacer conversión hacia arriba hacia una clase base, y en particular a una **interfaz**. (El efecto de producir una referencia a una interfaz desde un objeto que lo implementa es esencialmente el mismo que hacer una conversión hacia una clase base.) Esto se debe a que la clase interna —la implementación de la **interfaz**— puede ocultarse completamente y no estará disponible para nadie, lo cual es adecuado para ocultar la implementación. Todo lo que se logra a cambio es una referencia a la clase base o a la **interfaz**.

En primer lugar, se definirán las interfaces en sus propios archivos de forma que puedan ser usados en todos los ejemplos:

```
//: c08:Dentro.java
public interface Dentro {
    String leerEtiqueta();
} ///:~


//: c08:Contenidos.java
public interface Contenidos {
    int valor();
} ///:~
```

Ahora **Contenidos** y **Dentro** representan las interfaces disponibles para el programador cliente. (La **interfaz**, recuérdese, convierte sus miembros en **públicos** automáticamente.)

Cuando se obtiene de vuelta una referencia a la clase base o a la **interfaz**, es posible que se pueda incluso averiguar el tipo exacto, como se muestra a continuación:

```
//: c08:Paquete3.java
// Devolviendo una referencia a una clase interna.

public class Paquete3 {
  private class PContenidos implements Contenido {
    private int i = 11;
    public int valor() { return i; }
  }
  protected class PDestino
      implements Destino {
    private String etiqueta;
    private PDestino(String aDonde) {
      etiqueta = aDonde;
    }
    public String leerEtiqueta() { return etiqueta; }
  }
  public Destino dest(String s) {
    return new PDestino(s);
  }
  public Contenidos cont() {
    return new PContenidos();
  }
}

class Prueba {
  public static void main(String[] args) {
    Paquete3 p = new Paquete3();
    Contenidos c = p.cont();
    Destino d = p.dest("Tanzania");
    // Ilegal -- no se puede acceder a la clase privada:
    //! Paquete3.PContenidos pc = p.new PContenidos();
  }
} ///:~
```

Fíjese que, dado que **main( )** está en **Prueba**, si se desea ejecutar este programa no hay que ejecutar **Paquete3**, sino:

```
java Prueba
```

En el ejemplo, **main( )** debe estar en una clase separada para demostrar la privacidad de la clase interna **PContenidos**.

En **Paquete3**, se ha añadido algo nuevo: la clase interna **PContenidos** es **privada** de forma que nadie sino **Paquete3** puede acceder a ella. **PDestino** es **protegido**, por lo que nadie sino **Paquete3**, las clases contenidas en el paquete **Paquete3** (dado que **protegido** también permite acceso a nivel de paquete (es decir, protegido también es “amistoso”), y los herederos de **Paquete3**

pueden acceder a **PDestino**. Esto significa que el programador cliente tiene conocimiento y acceso restringidos a estos miembros. De hecho, no se puede hacer una conversión hacia abajo a una clase interna **privada** (o a una clase **protegida** interna a menos que se sea un descendiente), dado que no se puede acceder al nombre, como se puede ver en la **clase Prueba**. Por consiguiente, la clase interna **protegida** proporciona una forma para que el diseñador evite cualquier dependencia de codificación de tipos y oculte los detalles sobre implementación. Además, la extensión de una **interfaz** es inútil desde el punto de vista del programador cliente, dado que éste no puede acceder a ningún método adicional que no sea parte de la **interfaz pública** de la clase. Esto también proporciona una oportunidad para que el compilador Java genere código más eficiente.

Las clases normales (no internas) no pueden ser **privadas** o **protegidas** —sino sólo **públicas** o "amistosas".

Ámbitos y clases internas en métodos

Lo que se ha visto hasta la fecha abarca el uso típico de las clases internas. En general, el código que se escriba y lea relativo a las clases internas será clases internas "planas", simples y fáciles de entender. Sin embargo, el diseño de las clases internas es bastante completo y hay otras formas oscuras de usarlas: las clases internas pueden crearse dentro de un método o incluso en un ámbito arbitrario. Hay dos razones para hacer esto:

1. Como se ha visto previamente, se está implementando una interfaz de algún tipo, de forma que se puede crear y devolver una referencia.
2. Se está resolviendo un problema complicado y se desea crear una clase que ayude en la solución, aunque no se desea que ésta esté públicamente disponible.

En los ejemplos siguientes, se modificará el código anterior para utilizar:

1. Una clase definida dentro de un método.
2. Una clase definida dentro del ámbito de un método.
3. Una clase anónima que implementa una interfaz.
4. Una clase anónima que extienda una clase que no tenga un constructor por defecto.
5. Una clase anónima que lleve a cabo la inicialización de campos.
6. Una clase anónima que lleve a cabo la construcción usando inicialización de instancias (las clases internas anónimas no pueden tener constructores).

Aunque es una clase ordinaria con una implementación, también se usa **Envoltorio** como una "interfaz" común a sus clases derivadas:

```
//: c08:Envoltorio.java
public class Envoltorio {
  private int i;
```

```
  public Envoltorio(int x) { i = x; }
  public int valor() { return i; }
} ///:~
```

Se verá que **Envoltorio** tiene un constructor que necesita un parámetro, para hacer las cosas un poco más interesantes.

El primer ejemplo muestra la creación de una clase entera dentro del ámbito de un método (en vez de en el ámbito de otra clase):

```
//: c08:Paquete4.java
// Anidando una clase dentro de un método.

public class Paquete4 {
  public Destino dest(String s) {
    class PDestino
        implements Destino {
      private String etiqueta;
      private PDestino(String aDonde) {
        etiqueta = aDonde;
      }
      public String leerEtiqueta() { return etiqueta; }
    }
    return new PDestino(s);
  }
  public static void main(String[] args) {
    Paquete4 p = new Paquete4();
    Destino d = p.dest("Tanzania");
  }
} ///:~
```

La clase **Pdestino** es parte de **dest( )** más que de **Paquete4**. (Fíjese también que se podría usar el identificador de clase **PDestino** para una clase interna dentro de cada clase del mismo subdirectorio sin que haya colisión de nombres.) Por consiguiente, **PDestino** no puede ser accedida fuera del método **dest( )**. Fíjese que la conversión hacia arriba se da en la sentencia de retorno —nada viene de fuera de **dest( )** excepto una referencia a **PDestino**, la clase base. Por supuesto, el hecho de que el nombre de la clase **PDestino** se ubique dentro de **dest( )** no quiere decir que **PDestino** no sea un objeto válido una vez que **dest( )** devuelva su valor.

El siguiente ejemplo muestra cómo se puede anidar una clase interna dentro de cualquier ámbito arbitrario:

```
//: c08:Paquete5.java
// Anidando una clase dentro de un ámbito.

public class Paquete5 {
  private void rastreoInterno(boolean b) {
```

```
    if(b) {
      class RealizarRastreo {
        private String id;
        RealizarRastreo(String s) {
          id = s;
        }
        String obtenerId() { return id; }
      }
      RealizarRastreo ts = new RealizarRastreo("slip");
      String s = ts.obtenerId();
    }
    // ¡No se puede usar aquí, fuera del rango!
    //! RealizarRastreo ts = new RealizarRastreo("x");
  }
  public void rastrear() { rastreoInterno(true); }
  public static void main(String[] args) {
    Paquete5 p = new Paquete5();
    p.rastrear();
  }
} ///:~
```

La clase **RealizarRastreo** está anidada en el ámbito de una sentencia **if**. Esto no significa que la clase se cree condicionalmente —se compila junto con todo lo demás. Sin embargo, no está disponible fuera del rango en el que se definió. Por lo demás, tiene exactamente la misma apariencia que una clase ordinaria.

Clases internas anónimas

El siguiente ejemplo parece un poco extraño:

```
//: c08:Paquete6.java
// Un método que devuelve una clase interna anónima.

public class Paquete6 {
  public Contenidos cont() {
    return new Contenidos() {
      private int i = 11;
      public int valor() { return i; }
    }; // En este caso es necesario el punto y coma
  }
  public static void main(String[] args) {
    Paquete6 p = new Paquete6();
    Contenidos c = p.cont();
  }
} ///:~
```

¡El método **cont( )** combina la creación del valor de retorno con la definición de la clase que representa ese valor de retorno! Además, la clase es anónima —no tiene nombre. Para empeorar aún más las cosas, parece como si se estuviera empezando a crear un objeto **Contenidos**:

```
return new Contenidos()
```

Pero entonces, antes del punto y coma, se dice: "Pero espera, creo que me convertiré en una definición de clase":

```
return new Contenidos() {
    private int i = 11;
    public int valor() { return i; }
};
```

Lo que esta sintaxis significa es: "Crea un objeto de una clase anónima heredada de **Contenidos**". A la referencia que devuelva la expresión **new** se le hace un conversión hacia arriba automáticamente para convertirla en una referencia a **Contenidos**. La sintaxis de clase interna anónima es una abreviación de:

```
Class MisContenidos implements Contenidos {
  private int i = 11;
  public int valor() { return i; }
}
return new MisContenidos();
```

En la clase interna anónima, se crea **Contenidos** utilizando un constructor por defecto. El código siguiente muestra qué hacer si la clase base necesita un constructor con un argumento:

```
//: c08:Paquete7.java
// Una clase interna anónima que llama al
// constructor de la clase base.

public class Paquete7 {
  public Envoltorio envolver(int x) {
    // Llamada al constructor base:
    return new Envoltorio(x) {
      public int valor() {
        return super.valor() * 47;
      }
    }; // Punto y coma obligatorio
  }
  public static void main(String[] args) {
    Paquete7 p = new Paquete7();
    Envoltura w = p.envolver(10);
  }
} ///:~
```

Es decir, simplemente se pasa el argumento adecuado al constructor de la clase base, en este caso, se pasa **x** en **new Envoltorio(x)**. Una clase anónima no puede tener un constructor donde normalmente se invocaría a **super( )**.

En los dos ejemplos anteriores, el punto y coma no delimita el final del cuerpo de la clase, (como en C++). En cambio, marca el final de la expresión que viene a contener la clase anónima. Por consiguiente, es idéntico al uso de un punto y coma en cualquier otro sitio.

¿Qué ocurre si se necesita llevar a cabo algún tipo de inicialización de algún objeto de una clase interna anónima? Dado que es anónima, no se puede dar ningún nombre al constructor —por lo que no se puede tener un constructor. Se puede, sin embargo, llevar a cabo inicializaciones en el momento de definición de los campos:

```
//: c08:Paquete8.java
// Una clase interna anónima que lleva a cabo
// una inicialización. Versión abreviada de
// Paquete5.java.

public class Paquete8 {
  // El argumento debe ser constante para usarse en una
  // clase anónima interna:
  public Destino dest(final String dest) {
    return new Destino() {
      private String etiqueta = dest;
      public String leerEtiqueta() { return etiqueta; }
    };
  }
  public static void main(String[] args) {
    Paquete8 p = new Paquete8();
    Destino d = p.dest("Tanzania");
  }
} ///:~
```

Si se está definiendo una clase anónima interna y se desea utilizar un objeto definido fuera de la clase interna anónima, el compilador exige que el objeto externo sea **constante**. Ésta es la razón por la que el argumento pasado a **dest( )** es **constante**. Si se olvida, se obtendrá un mensaje de error en tiempo de compilación.

Mientras sólo se esté asignando un campo, el enfoque de arriba está bien. Pero ¿qué ocurre si se desea llevar a cabo alguna actividad al estilo constructor? Con la *inicialización de instancias*, se puede, en efecto, crear un constructor para una clase anónima interna.

```
//: c08:Paquete9.java
// Utilizando "inicialización de instancias" para llevar a cabo
// la construcción de una clase interna anónima.

public class Paquete9 {
```

```
  public Destino
  dest(final String dest, final float precio) {
    return new Destino() {
      private int coste;
      // Inicialización de instancias para cada objeto:
      {
        coste = Math.round(precio);
        if(cost > 100)
          System.out.println("¡Por encima del presupuesto!");
      }
      private String etiqueta = dest;
      public String leerEtiqueta() { return etiqueta; }
    };
  }
  public static void main(String[] args) {
    Paquete9 p = new Paquete9();
    Destino d = p.dest("Tanzania", 101.395F);
  }
} ///:~
```

Dentro del inicializador de instancias, se puede ver el código que no podría ser ejecutado como parte de un inicializador de campos (es decir, la sentencia **if**). Por tanto, en efecto, un inicializador de instancias es el constructor de una clase interna anónima. Por supuesto, está limitado; no se pueden sobrecargar inicializadores de instancias, por lo que sólo se puede tener uno de estos constructores.

El enlace con la clase externa

Hasta ahora, parece que las clases internas son solamente una ocultación de nombre y un esquema de organización de código, lo cual ayuda pero no convence. Sin embargo, hay otra alternativa. Cuando se crea una clase interna, un objeto a esa clase interna tiene un enlace al objeto contenedor que la hizo, y así puede acceder a los miembros del objeto contenedor —*sin* restricciones especiales. Además, las clases internas tienen derechos de acceso a todos los elementos de la clase contenedor[3]. El ejemplo siguiente lo demuestra:

```
//: c08:Secuencia.java
// Tiene una secuencia de objetos.

interface Selector {
  boolean fin();
  Object actual();
  void siguiente();
}
```

[3] Este enfoque varía mucho del diseño de las *clases anidadas* en C++, en el que estas clases son simplemente un mecanismo de ocultación de nombres. En C++, no hay ningún enlace al objeto contenedor ni permisos implícitos.

```
public class Secuencia {
  private Object[] obs;
  private int siguiente = 0;
  public Secuencia(int tamanio) {
    obs = new Object[tamanio];
  }
  public void aniadir(Object x) {
    if(siguiente < obs.length) {
      obs[siguiente] = x;
      siguiente++;
    }
  }
  private class SSelector implements Selector {
    int i = 0;
    public boolean fin() {
      return i == obs.length;
    }
    public Object actual() {
      return obs[i];
    }
    public void siguiente() {
      if(i < obs.length) i++;
    }
  }
  public Selector obtenerSelector() {
    return new SSelector();
  }
  public static void main(String[] args) {
    Secuencia s = new Secuencia(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.obtenerSelector();
    while(!sl.fin()) {
      System.out.println(sl.actual());
      sl.siguiente();
    }
  }
} ///:~
```

La **Secuencia** es simplemente un array de tamaño fijo de **Objetos** con una clase que lo envuelve. Para añadir un nuevo **Objeto** al final de la secuencia (si queda sitio) se llama a **aniadir( )**. Para buscar cada uno de los objetos de **Secuencia** hay una interfaz denominada **Selector** que permite ver si se está en el **fin( )**, echar un vistazo al método **actual( )**, y **siguiente( )** que permite moverse al siguiente objeto de la **Secuencia**. Dado que **Selector** es una **interfaz**, ésta puede ser implementada por otras muchas clases, y además los métodos podrían tomar la **interfaz** como parámetro, para crear código genérico.

Aquí, el **SSelector** es una clase **privada** que proporciona funcionalidad de **Selector**. En el método **main( )**, se puede ver la creación de una **Secuencia** seguida de la inserción de cierto número de objetos **String**. Después, se crea un **Selector** para llamar a **obtenerSelector( )** y éste se usa para moverse a través de la **Secuencia** y seleccionar cada elemento.

Al principio, la creación de **SSelector** parece simplemente otra clase interna. Pero examínela cuidadosamente. Fíjese que cada uno de los métodos **fin( )**, **actual( )**, y **siguiente( )** se refieren a **obs**, que es una referencia que no es parte de **SSelector**, sino un campo **privado** de la clase contenedora. Sin embargo, la clase interna puede acceder a métodos y campos de la clase contenedora como si les pertenecieran. Esto resulta ser muy conveniente, como se puede ver en el ejemplo de arriba.

Por tanto, una clase interna tiene acceso automático a los miembros de la clase contenedora. ¿Cómo puede ser esto? La clase interna debe mantener una referencia al objeto particular de la clase contenedora que era responsable de crearlo. Después, cuando se hace referencia al miembro de la clase contenedora, se usa esa referencia (oculta) para seleccionar ese miembro. Afortunadamente, el compilador se encarga de todos estos detalles, pero también podemos entender ahora que se pueda crear un objeto de una clase interna, sólo en asociación con un objeto de la clase contenedora. La construcción del objeto de la clase interna precisa de una referencia al objeto de la clase contenedora, y el compilador se quejará si no puede acceder a esa referencia. La mayoría de veces ocurre esto sin ninguna intervención por parte del programador.

Clases internas estáticas

Si no se necesita una conexión entre el objeto de la clase interna y el objeto de la clase externa, se puede hacer **estática** la clase interna. Para entender el significado de **estático** aplicado a clases internas, hay que recordar que el objeto de una clase interna ordinaria mantiene implícitamente una referencia al objeto de la clase contenedora que lo creó. Esto sin embargo no es cierto, cuando se dice que una clase interna es **estática**. Que una clase interna sea **estática** quiere decir que:

1. No se necesita un objeto de la clase externa para crear un objeto de una clase interna **estática**.

2. No se puede acceder a un objeto de una clase externa desde un objeto de una clase interna **estática**.

Las clases internas **estáticas** son distintas de las clases internas no **estáticas** también en otros aspectos. Los campos y métodos de las clases internas no **estáticas** sólo pueden estar en el nivel más externo de una clase, por lo que las clases internas no **estáticas** no pueden tener datos **estáticos**, campos **estáticos** o clases internas **estáticas**. Sin embargo, las clases internas **estáticas** pueden tener todo esto:

```
//: c08:Paquete10.java
// Clases internas estáticas.

public class Paquete10 {
  private static class PContenidos
  implements Contenidos {
    private int i = 11;
```

```
    public int valor() { return i; }
  }
  protected static class PDestino
      implements Destino {
    private String etiqueta;
    private PDestino(String aDonde) {
      etiqueta = aDonde;
    }
    public String leerEtiqueta() { return etiqueta; }
    // Las clases internas estáticas pueden tener
    // otros elementos estáticos:
    public static void f() {}
    static int x = 10;
    static class OtroNivel {
      public static void f() {}
      static int x = 10;
    }
  }
  public static Destino dest(String s) {
    return new PDestino(s);
  }
  public static Contenidos cont() {
    return new PContenidos();
  }
  public static void main(String[] args) {
    Contenidos c = cont();
    Destinos d = dest("Tanzania");
  }
} ///:~
```

En el método **main( )** no es necesario ningún objeto de **Paquete10**; en cambio, se usa la sintaxis normal para seleccionar un miembro **estático** para invocar a los métodos que devuelven referencias a **Contenidos** y **Destino**.

Como se verá en breve, en una clase interna ordinaria (no **estática**) se logra un enlace al objeto de la clase externa con una referencia especial **this**. Una clase interna **estática** no tiene esta referencia **this** especial, lo que la convierte en análoga a un método **estático**.

Normalmente no se puede poner código en una **interfaz**, pero una clase interna e**stática** puede ser parte de una **interfaz**. Dado que la clase es **estática** no viola las reglas de las interfaces —la clase interna **estática** sólo se ubica dentro del espacio de nombres de la interfaz:

```
//: c08:InterfazI.java
// Clases internas estáticas dentro de interfaces.

interface InterfazI {
```

```
    static class Interna {
      int i, j, k;
      public Interna() {}
      void f() {}
    }
} ///:~
```

Anteriormente sugerimos en este libro poner un método **main( )** en todas las clases para que actuara como banco de pruebas para cada una de ellas. Un inconveniente de esto es la cantidad de código compilado extra que se debe manejar. Si esto es un problema, se puede usar una clase interna **estática** para albergar el código de prueba:

```
//: c08:PruebaComa.java
// Poniendo código de pruebas en una clase estática interna.

class PruebaComa {
  PruebaComa() {}
  void f() { System.out.println("f()"); }
  public static class Probar {
    public static void main(String[] args) {
      PruebaComa t = new PruebaComa();
      t.f();
    }
  }
} ///:~
```

Esto genera una clase separada denominada **PruebaComa$Probar** (para ejecutar el programa, hay que decir **java PruebaComa$Probar**). Se puede usar esta clase para pruebas, pero no es necesario incluirla en el producto final.

Referirse al objeto de la clase externa

Si se necesita producir la referencia a la clase externa, se nombra la clase externa seguida por un punto y **this**. Por ejemplo, en la clase **Secuencia.SSelector**, cualquiera de sus métodos puede producir la referencia almacenada a la clase externa **Secuencia** diciendo **Secuencia.this**. La referencia resultante es automáticamente del tipo correcto. (Esto se conoce y comprueba en tiempo de compilación, por lo que no hay sobrecarga en tiempo de ejecución.)

En ocasiones, se desea decir a algún otro objeto que cree un objeto de una de sus clases internas. Para hacer esto hay que proporcionar una referencia al objeto de la otra clase externa en la expresión **new**, como en:

```
//: c08:Paquete11.java
// Creando instancias de clases internas.

public class Paquete11 {
```

```
  class Contenidos {
    private int i = 11;
    public int valor()() { return i; }
  }
  class Destino {
    private String etiqueta;
    Destino(String aDonde) {
      etiqueta = aDonde;
    }
    String leerEtiqueta() { return etiqueta; }
  }
  public static void main(String[] args) {
    Paquete11 p = new Paquete11();
    // Debe usar instancia de la clase externa
    // para crear una instancia de la clase interna:
    Paquete11.Contenidos c = p.new Contenidos();
    Paquete11.Destino d =
      p.new Destino("Tanzania");
  }
} ///:~
```

Para crear un objeto de la clase interna directamente, no se obra igual refiriéndose a la clase externa **Paquete11** como cabría esperar, sino que se une un *objeto* de la clase externa para construir un objeto de la clase interna:

```
Paquete11.Contenidos c = p.new Contenidos();
```

Por tanto, no es posible crear un objeto de la clase interna a menos que ya se tenga un objeto de la case externa. Esto se debe a que el objeto de la clase interna está conectado al objeto de la clase externa del que fue hecho. Sin embargo, si se hace una clase interna **estática**, entonces no es necesaria una referencia al objeto de la clase externa.

Acceso desde una clase múltiplemente anidada

No importa lo profundo que se anide una clase interna —puede acceder transparentemente a todos los miembros de todas las clases en los que esté anidada, como se muestra aquí:[3]

```
//: c08:AccesoAnidamientoMultiple.java
// Las clases anidadas pueden acceder a todos los miembros de todos
// niveles de las clases en las que están anidadas.

class AAM {
```

[3] Gracias de nuevo a Martin Danner.

```
  private void f() {}
  class A {
    private void g() {}
    public class B {
      void h() {
        g();
        f();
      }
    }
  }
}

public class AccesoAnidamientoMultiple {
  public static void main(String[] args) {
    AAM aam = new AAM();
    AAM.A aama = aam.new A();
    AAM.A.B aamab = aama.new B();
    aamab.h();
  }
} ///:~
```

Se puede ver que en **AAM.A.B**, los métodos **g( )** y **f( )** pueden ser invocados sin ningún tipo de restricción (a pesar de que sean **privados**). Este ejemplo también demuestra la sintaxis necesaria para crear objetos de clases internas múltiplemente anidadas cuando se crean los objetos en una clase distinta. La sintaxis "**.new**" produce el ámbito correcto por lo que no es necesario restringir el nombre de la clase en la llamada al constructor.

Heredar de clases internas

Dado que hay que adjuntar el constructor de la clase interna a la referencia del objeto de la clase contenedora, las cosas son ligeramente complicadas cuando se trata de heredar de una clase interna. El problema es que se debe inicializar la referencia "secreta" al objeto contenedor, y además en la clase derivada deja de haber un objeto por defecto al que adjuntarla. La respuesta es usar una sintaxis propuesta para hacer la asociación explícita:

```
//: c08:HerenciaInterna.java
// Heredando una clase interna.

class ConInterna {
  class Interna {}
}

public class HerenciaInterna
    extends HerenciaInterna.Interna {
  //! HerenciaInterna() {} // No compila
```

```
   HerenciaInterna(ConInterna wi) {
      wi.super();
   }
   public static void main(String[] args) {
      ConInterna wi = new ConInterna();
      HerenciaInterna ii = new HerenciaInterna(wi);
   }
} ///:~
```

Se puede ver que **HerenciaInterna** sólo está extendiendo la clase interna, y no la externa. Pero cuando llega la hora de crear un constructor, el constructor por defecto no es bueno y no se puede simplemente pasar una referencia a un objeto contenedor. Además, hay que usar la sintaxis:

```
referenciaClaseContenedora.super();
```

dentro del constructor. Esto proporciona la referencia necesaria para que el programa compile.

¿Pueden superponerse las clases internas?

¿Qué ocurre cuando se crea una clase interna, se hereda de la clase contenedora y se redefine la clase interna? Es decir, ¿es posible superponer una clase interna? Esto sería un concepto poderoso, pero la "superposición" en una clase interna como si fuera otro método de la clase externa verdaderamente no sirve para nada:

```
//: c08:HuevoGrande.java
// Una clase interna no se superpone como un método.

class Huevo {
   protected class Yema {
      public Yema() {
         System.out.println("Huevo.Yema()");
      }
   }
   private Yema y;
   public Huevo() {
      System.out.println("New Huevo()");
      y = new Yema();
   }
}

public class HuevoGrande extends Huevo {
   public class Yema {
      public Yema() {
         System.out.println("HuevoGrande.Yema()");
      }
   }
```

```
  public static void main(String[] args) {
    new HuevoGrande();
  }
} ///:~
```

El compilador crea automáticamente el constructor por defecto, y éste llama al constructor por defecto de la clase base. Se podría pensar, que dado que se está creando **HuevoGrande**, debería usarse la versión "superpuesta" de **Yema**, pero éste no es el caso. La salida es:

```
New_Huevo()
Huevo.Yema()
```

Este ejemplo simplemente muestra que no hay ninguna magia extra propia de la clase interna cuando se hereda desde la clase externa. Las dos clases internas constituyen entidades completamente separadas, cada una en su propio espacio de nombres. Sin embargo, sigue siendo posible heredar explícitamente desde la clase interna:

```
//: c08:HuevoGrande2.java
// Herencia correcta de una clase interna.

class Huevo2 {
  protected class Yema {
    public Yema() {
      System.out.println("Huevo2.Yema()");
    }
    public void f() {
      System.out.println("Huevo2.Yema.f()");
    }
  }
  private Yema y = new Yema();
  public Huevo2() {
    System.out.println("New Huevo2()");
  }
  public void insertarYema(Yema yy) { y = yy; }
  public void g() { y.f(); }
}

public class HuevoGrande2 extends Huevo2 {
  public class Yema extends Huevo2.Yema {
    public Yema() {
      System.out.println("HuevoGrande2.Yema()");
    }
    public void f() {
      System.out.println("HuevoGrande2.Yema.f()");
    }
  }
```

```
  public HuevoGrande2() { insertaYema(new Yema()); }
  public static void main(String[] args) {
    Huevo2 e2 = new HuevoGrande2();
    e2.g();
  }
} ///:~
```

Ahora **HuevoGrande2.Yema** hereda explícitamente de **Huevo2.Yema** y superpone sus métodos. El método **insertaYema( )** permite a **HuevoGrande2** hacer una conversión hacia arriba a uno de sus propios objetos **Yema** hacia la referencia **y** de **Huevo2**, de forma que cuando **g( )** llama a **y.f( )** se usa la versión superpuesta de **f( )**. La salida es:

```
Huevo2.Yema()
New Huevo2()
Huevo2.Yema()
HuevoGrande2.Yema()
HuevoGrande2.Yema.f()
```

La segunda llamada a **Huevo2.Yema( )** es la llamada al constructor de la clase base del constructor **HuevoGrande2.Yema**. Se puede ver que se usa la versión superpuesta de **f( )** al llamar a **g( )**.

Identificadores de clases internas

Dado que toda clase produce un archivo **.class** que mantiene toda la información de como crear objetos de ese tipo (esta información produce una "meta-clase" llamada objeto **Class**), se podría adivinar que también las clases internas deben producir archivos **.class** para contener la información de *sus* objetos **Class**. Los nombres de estos archivos/clases tienen una fórmula estricta: el nombre de la clase contenedora, seguida de un "$", seguida del nombre de la clase interna. Por ejemplo, los ficheros **.class** creados por **HerenciaInterna.java** incluyen:

```
HerenciaInterna.class
ConInterna$Interna.class
ConInterna.class
```

Si las clases internas son anónimas, el compilador simplemente genera números como identificadores de las clases internas. Si las clases internas están anidadas dentro de clases internas, sus nombres simplemente se añaden tras un **"$"** y los identificadores de la clase externa.

Aunque este esquema de generación de nombres internos es simple y directo, también es robusto y maneja la mayoría de situaciones[5]. Dado que éste es el esquema de nombres estándar de Java, los ficheros generados son directamente independientes de la plataforma. (Nótese que el compilador de Java cambia las clases internas hasta hacerlas funcionar.)

---

[5] Por otro lado, "$" es un metacarácter para el *shell* de Unix, por lo que en ocasiones habrá problemas para listar las clases **.class**. Esto es un poco extraño viniendo de Sun, una compañía basada en Unix. Adivinamos que no tuvieron este aspecto en cuenta, y sin embargo, pensarían que había que centrarse en los ficheros de código fuente.

¿Por qué clases internas?

Hasta ahora se ha visto mucha sintaxis y semántica que describen el funcionamiento de las clases internas, pero esto no contesta a la pregunta de por qué existen. ¿Por qué Sun se metió en tanto lío para añadir esta característica fundamental del lenguaje?

Generalmente, la clase interna hereda de una clase o implementa una **interfaz**, y el código de la clase interna manipula el objeto de la clase externa en la que se ha creado. Por tanto, se podría decir que una clase interna proporciona una especie de ventana dentro de la clase externa.

Una pregunta que llega al corazón de las clases internas es: si simplemente se necesita una referencia a una **interfaz** ¿por qué no hacer simplemente que la clase externa implemente esa **interfaz**? La respuesta es: "Si eso es todo lo que necesitas, entonces así deberías hacerlo". Entonces, ¿qué es lo que distingue una clase interna que implementa una **interfaz** de una clase externa que implemente la misma **interfaz**? La respuesta es que siempre se puede tener la comodidad de las **interfaces** —algunas veces se trabaja con implementaciones. Por tanto, la razón más convincente para las clases internas es:

> *Cada clase interna puede heredar independientemente de una implementación. Por consiguiente, la clase interna no está limitada por el hecho de que la clase externa pueda estar ya heredando de una implementación.*

Sin la habilidad quc las clases internas proporcionan para heredar —de hecho— desde más de una clase concreta o **abstracta**, algunos diseños y problemas de programación serían intratables. Por tanto, una forma de mirar a la clase interna es como la terminación de la solución del problema de la herencia múltiple. Las interfaces solucionan parte del problema, pero las clases internas permiten la "herencia de implementación múltiple" de manera efectiva. Es decir, las clases internas permiten heredar de forma efectiva de más de otro no-**interfaz**.

Para ver esto con mayor detalle, considérese una situación en que se tengan dos interfaces que de alguna manera deban implementarse dentro de una clase. Debido a la flexibilidad de las interfaces, se tienen dos alternativas: una única clase o una clase interna:

```
//: c08:InterfacesMultiples.java
// Dos formas en las que una clase puede
// implementar múltiples interfaces.

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
  B creaB() {
    // Clase interna anónima:
    return new B() {};
  }
```

```
}

public class InterfacesMultiples {
  static void tomaA(A a) {}
  static void tomaB(B b) {}
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    tomaA(x);
    tomaA(y);
    tomaB(x);
    tomaB(y.creaB());
  }
} ///:~
```

Por supuesto, esto asume que la estructura del código tiene sentido de alguna forma. Sin embargo, generalmente se tendrá algún tipo de guía, en la propia naturaleza del problema, sobre si usar una clase o una clase interna. Pero sin más restricciones, en el ejemplo de arriba el enfoque que se sigue no es muy diferente desde el punto de vista de la implementación. Ambos funcionan.

Sin embargo, si se tienen clases **abstractas** o concretas en vez de **interfaces**, nos limitamos repentinamente a usar clases internas si la clase debe implementar de alguna manera las otras dos:

```
//: c08:ImplementacionesMultiples.java
// Con clases concretas o abstractas, las clases
// internas son la única manera de producir el efecto de
// "herencia de implementación múltiple".

class C {}
abstract class D {}

class Z extends C {
  D crearD() { return new D() {}; }
}

public class ImplementacionesMultiples {
  static void tomarC(C c) {}
  static void tomarD(D d) {}
  public static void main(String[] args) {
    Z z = new Z();
    tomarC(z);
    tomarD(z.crearD());
  }
} ///:~
```

Si no se deseara resolver el problema de la "herencia de implementación múltiple", se podría codificar posiblemente todo lo demás sin la necesidad de clases internas. Pero con las clases internas se tienen estas características adicionales:

1. La clase interna tiene múltiples instancias, cada una con su propia información de estado que es independiente de la información del objeto de la clase externa.
2. En una clase externa se pueden tener varias clases internas, cada una de las cuales implementa la misma **interfaz** o hereda de la misma clase de distinta forma. En breve se mostrará un ejemplo de esto.
3. El momento de creación del objeto de la clase interna no está atado a la creación del objeto de la clase externa.
4. No hay relaciones "es-un" potencialmente confusas dentro de la clase interna; se trata de una entidad separada.

Como ejemplo, si **Secuencia.java** no usara clases internas, habría que decir que "una **Secuencia** es un **Selector**", y sólo se podría tener un **Selector** para una **Secuencia** particular. Además, se puede tener un segundo método, **obtenerRSelector( )**, que produce un **Selector** que se mueve hacia atrás por la secuencia. Este tipo de flexibilidad sólo está disponible con clases internas.

Cierres (closures) y Retrollamadas (Callbacks)

Un *cierre* es un objeto invocable que retiene información del ámbito en el que se creó. Por definición, se puede ver que una clase interna es un *cierre* orientado a objetos, porque no se limita a contener cada fragmento de información del objeto de la clase externa ("el ámbito en el que ha sido creada"), sino que mantiene automáticamente una referencia de vuelta al objeto completo de la clase externa, en el que tiene permiso para manipular todos los miembros, incluso los **privados**.

Uno de los argumentos más convincentes, hechos para incluir algún tipo de mecanismo apuntador en Java, era permitir las llamadas hacia atrás o *retrollamadas*. Con una *retrollamada*, se da a otro objeto de java un fragmento de información que le permite invocar al objeto que lo originó en algún momento. Éste es un concepto muy potente, como se verá en los Capítulos 13 y 16. Si se implementa una retrollamada utilizando un puntero, sin embargo, hay que confiar en que el programador se comporte adecuadamente y no use incorrectamente el puntero. Como se ha visto hasta ahora, Java tiende a ser más cuidadoso, de forma que no se incluyen punteros en el propio lenguaje.

El cierre proporcionado por la clase interna es la solución perfecta; más flexible y mucho más segura que un puntero. He aquí un ejemplo simple:

```
//: c08:Retrollamadas.java
// Utilizando clases internas para retrollamadas

interface Incrementable {
  void incrementar();
}
```

```
// Muy simple para simplemente implementar la interfaz:
class Llamada1 implements Incrementable {
  private int i = 0;
  public void incrementar() {
    i++;
    System.out.println(i);
  }
}

class MiIncremento {
  public void incremento() {
    System.out.println("Otra operacion");
  }
  public static void f(MiIncremento mi) {
    mi.incrementar();
  }
}

// Si tu clase debe implementar incrementar() de alguna u otra
// manera, hay que usar una clase interna:
class Llamada2 extends MiIncremento {
  private int i = 0;
  private void incr() {
    i++;
    System.out.println(i);
  }
  private class Cierre implements Incrementable {
    public void incrementar() { incr(); }
  }
  Incrementable obtenerReferenciaRetrollamada() {
    return new Cierre();
  }
}

class Visita {
  private Incrementable referenciaRetrollamada;
  Visita(Incrementable cbh) {
    referenciaRetrollamada = cbh;
  }
  void realizar() {
    referenciaRetrollamada.incrementar();
  }
}

public class Retrollamada {
```

```
   public static void main(String[] args) {
      Llamada1 c1 = new Llamada1();
      Llamada2 c2 = new Llamada2();
      MiIncremento.f(c2);
      Visita visita1 = new Visita(c1);
      Visita visita2 =
         new Visita(c2.obtenerReferenciaRetrollamada());
      visita1.realizar();
      visita1.realizar();
      visita2.realizar();
      visita2.realizar();
   }
} ///:~
```

Este ejemplo también proporciona una distinción aún mayor entre implementar una interfaz en una clase externa o hacerlo en una interna. **Llamada1** es claramente la solución más simple en lo que a código se refiere. **Llamada2** hereda de **MiIncremento**, que ya tiene un método **incrementar( )** diferente que hace algo no relacionado con lo que se espera de la interfaz **Incrementable**. Cuando se hereda **Llamada2.incrementar** de **MiIncremento**, no se puede superponer **incrementar( )** para ser usado por parte de **Incrementable**, por lo que uno se ve forzado a proporcionar una implementación separada utilizando una clase interna. Fíjese también que cuando se crea una clase interna no se añade o modifica la interfaz de la clase externa.

Téngase en cuenta que en **Llamada2** todo menos **obtenerReferenciaRetrollamada( )** es **privado**. Para permitir *cualquier* conexión al mundo exterior, es esencial la **interfaz Incrementable**. Aquí se puede ver cómo las **interface**s permiten una completa separación de la interfaz de la implementación.

La clase interna **Cierre** simplemente implementa **Implementable** para proporcionar un anzuelo de vuelta a **Llamada2** —pero un anzuelo seguro. Quien logre una referencia **Incrementable** puede, por supuesto, invocar sólo a **incrementar( )** y no tiene otras posibilidades (a diferencia de un puntero, que permitiría acceso total).

Visita toma una referencia **Incrementable** en su constructor (aunque la captura de la referencia a la retrollamada podría darse en cualquier momento) y entonces, algo después, utiliza la referencia para hacer una "llamada hacia atrás" a la clase **Llamada**.

El valor de una retrollamada reside en su flexibilidad —se puede decidir dinámicamente qué funciones serán invocadas en tiempo de ejecución. El beneficio de esto se mostrará más evidentemente en el Capítulo 13, en el que se usan retrollamadas en todas partes para implementar la funcionalidad de la interfaz gráfica de usuario (IGU).

Clases internas y sistema de control

Un ejemplo más concreto del uso de las clases internas sería lo que llamamos *sistema de control.*

Un *sistema de aplicación* es una clase o conjunto de clases diseñado para solucionar un tipo particular de problema. Para aplicar un sistema de aplicación, se hereda de una o más clases y se su-

perponen algunos de los métodos. El código que se escribe en los métodos superpuestos particulariza la solución proporcionada por el sistema de aplicación, para solucionar un problema específico. El sistema de control es un tipo particular de sistema de aplicación dominado por la necesidad de responder a eventos; un sistema que responde principalmente a eventos se denomina *sistema dirigido por eventos*. Uno de los problemas más importantes en la programación de aplicaciones es la interfaz gráfica de usuario (IGU), que está dirigida a eventos casi completamente. Como se verá en el Capítulo 13, la biblioteca Swing de Java es un sistema de control que soluciona el problema del IGU de forma elegante, utilizando intensivamente clases internas.

Para ver cómo las clases internas permiten la creación y uso de forma simple de sistemas de control, considérese uno cuyo trabajo es ejecutar eventos siempre que éstos estén "listos". Aunque "listos" podría significar cualquier caso, en este caso se usará su significado por defecto, basado en el reloj. Lo que sigue es un sistema de control que no contiene información específica sobre qué se está controlando. En primer lugar, he aquí una interfaz que describe cualquier evento de control. Es una clase **abstracta** en vez de una **interfaz** porque su comportamiento por defecto es llevar a cabo el control basado en el tiempo, por lo que se puede incluir ya alguna implementación:

```
//: c08:controlador:Evento.java
// Los métodos comunes para cualquier evento de control.
package c08.controlador;

abstract public class Evento {
  private long instEvento;
  public Evento(long instanteEvento) {
    instEvento = instanteEvento;
  }
  public boolean listo() {
    return System.currentTimeMillis() >= instEvento;
  }
  abstract public void accion();
  abstract public String descripcion();
} ///:~
```

El constructor simplemente captura el instante de tiempo en el que que se desea que se ejecute el **Evento**, mientras que **listo( )** dice cuándo es hora de ejecutarlo. Por supuesto, podría superponerse **listo( )** en alguna clase derivada de la clase **Descripción**.

El método **accion( )** es el que se invoca cuando el **Evento** está **listo( )**, y **descripción** da información textual sobre el **Evento**.

El siguiente fichero contiene el sistema de control que gestiona eventos de incendios. La primera clase es realmente una clase "ayudante" cuyo trabajo es albergar objetos **Evento**. Se puede sustituir por cualquier contenedor apropiado, y en el Capítulo 9 se descubrirán otros contenedores que proporcionarán este truco sin exigir la escritura de código extra:

```
//: c08:controlador:Controlador.java
// Junto con Evento, el sistema genérico
// para todos los sistemas de control:
```

```
package c08.controlador;

// Esto es simplemente una forma de guardar objetos Evento.
class ConjuntoEventos {
  private Evento[] eventos = new Evento[100];
  private int indice = 0;
  private int siguiente = 0;
  public void aniadir(Evento e) {
    if(indice >= eventos.length)
      return; // (En realidad, lanzar una excepción)
    eventos[indice++] = e;
  }
  public Evento obtenerSiguiente() {
    boolean vuelta = false;
    int primero = siguiente;
    do {
      siguiente = (siguiente + 1) % eventos.length;
      // Ver si ha vuelto al principio:
      if(primero == siguiente) vuelta = true;
      // Si va más allá de primero, la lista está vacía:
      if((siguiente == (primero + 1) % eventos.length)
         && eventos)
        return null;
    } while(eventos[siguiente] == null);
    return eventos[siguiente];
  }
  public void eliminarActual() {
    eventos[siguiente] = null;
  }
}

public class Controlador {
  private ConjuntoEventos es = new ConjuntoEventos();
  public void aniadirEvento(Evento c) { es.aniadir(c); }
  public void ejecutar() {
    Evento e;
    while((e = es.obtenerSiguiente()) != null) {
      if(e.listo()) {
        e.accion();
        System.out.println(e.descripcion());
        es.eliminarActual();
      }
    }
  }
} ///:~
```

ConjuntoEventos mantiene arbitrariamente 100 **objetos de tipo Evento**. (Si se usara un contenedor de los que veremos en el Capítulo 9 no haría falta preocuparse por su tamaño máximo, puesto que se recalcularía por sí mismo.) El **índice** se usa para mantener información del espacio disponible, y **siguiente** se usa cuando se busca el siguiente **Evento** de la lista, para ver si ya se ha dado la vuelta o no. Esto es importante durante una llamada a **obtenerSiguiente( )**, porque los objetos **Evento** se van eliminando de la lista (utilizando **eliminarActual( )**) una vez que se ejecutan, por lo que **obtenerSiguiente( )** encontrará espacios libres en la lista al recorrerla.

Nótese que **eliminarActual( )** no se dedica simplemente a poner algún indicador que muestre que el objeto ya no está en uso. En vez de esto, pone la referencia a **null**. Esto es importante porque si el recolector de basura ve una referencia que sigue estando en uso, no puede limpiar el objeto. Si uno piensa que sus referencias podrían quedarse colgadas (como ocurre aquí), es buena idea ponerlas a **null** para dar permiso al recolector de basura y que los limpie.

Es en **Controlador** donde se da el verdadero trabajo. Utiliza un **ConjuntoEventos** para mantener sus objetos **Evento**, y **aniadirEvento( )** permite añadir nuevos eventos a esta lista. Pero el método importante es **ejecutar( )**. Este método se mete en un bucle en **ConjuntoEventos**, buscando un objeto **Evento** que esté **listo( )** para ser ejecutado. Por cada uno que encuentre **listo( )**, llama al método **accion( )**, imprime la **descripcion( )**, y quita el **Evento** de la lista.

Fíjese que hasta ahora en este diseño no se sabe nada de *qué* hace un **Evento**. Y esto es lo esencial del diseño; cómo "separa las cosas que cambian de las que permanecen igual". O, usando el término, el "vector de cambio" está formado por las diferentes acciones de varios tipos de objetos **Evento**, y uno expresa acciones diferentes creando distintas subclases **Evento**.

Es aquí donde intervienen las clases internas, que permiten dos cosas:

1. Crear la implementación completa de una aplicación de sistema de control en una única clase, encapsulando, por tanto, todo lo que sea único de la implementación. Se usan las clases internas para expresar los distintos tipos de **accion( )** necesarios para resolver el problema. Además, el ejemplo siguiente usa clases internas **privadas**, por lo que la implementación está completamente oculta y puede ser cambiada con impunidad.

2. Las clases internas hacen que esta implementación no sea excesivamente complicada porque se puede acceder sencillamente a cualquiera de los miembros de la clase exterior. Sin esta capacidad, el código podría volverse tan incómodo de manejar que se acabaría buscando otra alternativa.

Considérese una implementación particular del sistema de control diseñada para controlar las funciones de un invernadero[6]. Cada acción es completamente distinta: encender y apagar las luces, agua y termostatos, hacer sonar timbres, y reinicializar el sistema. Pero el sistema de control está diseñado para aislar fácilmente este código diferente. Las clases internas permiten tener múltiples versiones derivadas de la misma clase base, **Evento**, dentro de una única clase. Por cada tipo de acción se hereda una nueva clase interna **Evento**, y se escribe el código de control dentro de **accion( )**.

---

[6] Por algún motivo, éste siempre ha sido un problema que nos ha gustado resolver; viene en el libro *C++ Inside & Out*, pero Java permite una solución mucho más elegante.

Como es típico con un sistema de aplicación, la clase **ControlesInvernadero** se hereda de **Controlador**:

```
//: c08:ControlesInvernadero.java
// Aplicación especifica del sistema de
// control, toda ella en una única clase. Las clases internas
// permiten encapsular diferentes funcionalidades
// por cada tipo de evento.
import c08.controlador.*;

public class ControlesInvernadero
    extends Controlador {
  private boolean luz = false;
  private boolean agua = false;
  private String termostato = "Dia";
  private class EncenderLuz extends Evento {
    public EncenderLuz(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      // para encender físicamente la luz.
      luz = true;
    }
    public String descripcion() {
      return "Luz encencida";
    }
  }
  private class ApagarLuz extends Evento {
    public ApagarLuz(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      // para apagar físicamente la luz.
      luz = false;
    }
    public String descripcion() {
      return "Luz apagada";
    }
  }
  private class EncenderAgua extends Evento {
    public EncenderAgua(long instanteEvento) {
      super(instanteEvento);
    }
```

```
    public void accion() {
      // Poner aquí el código de control de hardware
      agua = true;
    }
    public String descripcion() {
      return "Agua del invernadero encendida";
    }
  }
  private class ApagarAgua extends Evento {
    public ApagarAgua(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      agua = false;
    }
    public String descripcion() {
      return "Agua del invernadero apagada";
    }
  }
  private class TermostatoNoche extends Evento {
    public TermostatoNoche(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      termostato = "Noche";
    }
    public String descripcion() {
      return "Termostato activado para la noche";
    }
  }
  private class TermostatoDia extends Evento {
    public TermostatoDia(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      termostato = "Dia";
    }
    public String descripcion() {
      return "Termostato activado para el dia";
    }
  }
  // Ejemplo de una acción() que inserta una
```

```
// nueva acción dentro de la lista de eventos:
private int timbres;
private class Campana extends Evento {
   public Campana(long instanteEvento) {
      super(instanteEvento);
   }
   public void accion() {
      // Sonar cada 2 segundos, 'timbres':
      System.out.println("¡Ring!");
      if(--timbres > 0)
         aniadirEvento(new Campana(
            System.currentTimeMillis() + 2000));
   }
   public String descripcion() {
      return "Sonar la campana";
   }
}
private class Rearrancar extends Evento {
   public rearrancar(long instanteEvento) {
      super(instanteEvento);
   }
   public void accion() {
      long tm = System.currentTimeMillis();
      // En vez de cableado se podría poner información
      // de configuración de un fichero de texto aquí:
      timbres = 5;
      aniadirEvento(new TermostatoNoche(tm));
      aniadirEvento(new EncenderLuz(tm + 1000));
      aniadirEvento(new ApagarLUz(tm + 2000));
      aniadirEvento(new EncenderAgua(tm + 3000));
      aniadirEvento(new ApagarAgua(tm + 8000));
      aniadirEvento(new Campana(tm + 9000));
      aniadirEvento(new TermostatoDia(tm + 10000));
      // ¡Incluso se puede añadir un objeto rearrancar!
      aniadirEvento(new Rearrancar(tm + 20000));
   }
   public String descripcion() {
      return "Reiniciando el sistema";
   }
}
public static void main(String[] args) {
   ControlesInvernadero gc =
      new ControlesInvernadero();
   long tm = System.currentTimeMillis();
   gc.aniadirEvento(gc.new Rearrancar(tm));
```

```
        gc.ejecutar();
    }
} ///:~
```

Fíjese que **luz**, **agua**, **termostato** y **campanas** pertenecen a la clase externa **ControlesInvernadero**, y sin embargo las clases internas pueden acceder a esos campos sin restricciones o permisos especiales. Además, la mayoría de los métodos **accion( )** implican algún tipo de control hardware, que con mucha probabilidad incluirán llamadas a código no Java.

La mayoría de clases **Evento** tienen la misma apariencia, pero **Campana** y **Rearrancar** son especiales. La **Campana** suena, y si no ha sonado aún el número suficiente de veces, añade un nuevo objeto **Campana** a la lista de eventos, de forma que volverá a sonar más tarde. Téngase en cuenta cómo las clases internas parecen una herencia múltiple: **Campana** tiene todos los métodos de **Evento** y también parece tener todos los métodos de la clase externa **ControlesInvernadero**.

Rearrancar es la responsable de inicializar el sistema, de forma que añade todos los eventos apropiados. Por supuesto, se puede lograr lo mismo de forma más flexible evitando codificar los eventos y en vez de ello leerlos de un archivo. (Un ejercicio que se pide en el Capítulo 11 es modificar este ejemplo para que haga justamente eso.) Dado que **Rearrancar( )** es simplemente otro objeto **Evento( )**, se puede añadir también un objeto **Rearrancar** dentro de **Rearrancar.accion( )**, de forma que el sistema se inicie a sí mismo regularmente. Y todo lo que se necesita hacer en el método **main( )** es crear un objeto **ControlesInvernadero** y añadir un objeto **Rearrancar** para que funcione.

Este ejemplo debería transportar al lector un gran paso hacia adelante al apreciar el valor de las clases internas, especialmente cuando se usan dentro de un sistema de control. Sin embargo, en el Capítulo 13 se verá cómo se usan estas clases elegantemente para describir las acciones de una interfaz gráfica de usuario. Para cuando se acabe ese capítulo, todo lector sabrá manejarlos.

Resumen

Las interfaces y las clases internas son conceptos más sofisticados que lo que se encontrará en la mayoría de lenguajes de POO. Por ejemplo, no hay nada igual en C++. Juntos, solucionan el mismo problema que C++ trata de solucionar con su característica de herencia múltiple[6]. Sin embargo, la MI de C++ resulta ser bastante difícil de usar, mientras que las clases internas e interfaces de Java son, en comparación, mucho más accesibles.

Aunque las características por sí mismas son bastante directas, usarlas es un aspecto de diseño, al igual que ocurre con el polimorfismo. Con el tiempo, uno será capaz de reconocer mejor las situaciones en las que hay que usar una interfaz, o una clase interna, o ambas. Pero en este punto del libro, deberíamos habernos familiarizado con su sintaxis y semántica. A medida que se vea el uso de estas características, uno las irá haciendo propias.

---

[6] N. del traductor: *Múltiple Inheritance* (MI) en C++.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com.*

1. Probar que los métodos de una **interfaz** son implícitamente **estáticos** y **constantes**.
2. Crear una **interfaz** que contenga tres métodos en su propio **paquete**. Implementar la interfaz en un **paquete** diferente.
3. Probar que todos los métodos de una **interfaz** son automáticamente **públicos**.
4. En **c07:Bocadillo.java**, crear una interfaz denominada **ComidaRapida** (con métodos apropiados) y cambiar **Bocadillo** de forma que también implemente **ComidaRapida**.
5. Crear tres **interfaces**, cada uno con dos métodos. Heredar una nueva **interfaz** de las tres, añadiendo un nuevo método. Crear una clase implementando la nueva **interfaz** y heredando además de una clase concreta. Ahora escribir cuatro métodos, cada uno de los cuales toma una de las cuatro **interfaces** como parámetro. En el método **main( )**, crear un objeto de la nueva clase y pasárselo a cada uno de los métodos.
6. Modificar el Ejercicio 5, creando una clase **abstracta** y heredándola en la clase derivada.
7. Modificar **Musica5.java** añadiendo una **interfaz Tocable**. Eliminar la declaración **tocar( )** de **Instrumento**. Añadir **Tocable** a las clases derivadas incluyéndolo en la lista de interfaces que **implementa**. Cambiar **afinar( )** de forma que tome un **Tocable** en vez de un **Instrumento**.
8. Cambiar el Ejercicio 6 del Capítulo 7 de forma que **Roedor** sea una **interfaz**.
9. En **Aventura.java**, añadir una **interfaz** denominada **PuedeTrepar** siguiendo el modelo de las otras interfaces.
10. Escribir un programa que importe y use **Mes2.java**.
11. Siguiendo el ejemplo de **Mes2.java**, crear una enumeración de los días de la semana.
12. Crear una **interfaz** con al menos un método, en su propio paquete. Crear una clase en otro paquete. Añadir una clase interna **protegida** que implemente la **interfaz**. En un tercer paquete, heredar de la nueva clase y, dentro de un método, devolver un objeto de la clase interna **protegida**, haciendo un conversión hacia arriba a la **interfaz** durante este retorno.
13. Crear una **interfaz** con al menos un método, e implementar esa **interfaz** definiendo una clase interna dentro de un método, que devuelva una referencia a la **interfaz**.
14. Repetir el Ejercicio 13, pero definir la clase interna dentro del ámbito de un método.
15. Repetir el Ejercicio 13 utilizando una clase interna anónima.
16. Crear una clase interna **privada** que implemente una **interfaz pública**. Escribir un método que devuelva una referencia a una instancia de la clase interna **privada**, hacer un conversión

hacia arriba a la **interfaz**. Mostrar que la clase interna está totalmente oculta intentando hacer una, conversión hacia abajo de la misma.

17. Crear una clase con un constructor distinto del constructor por defecto, y sin éste. Crear una segunda clase que tenga un método que devuelva una referencia a la primera clase. Crear el objeto a devolver haciendo una clase interna anónima que herede de la primera clase.

18. Crear una clase con un atributo **privado** y un método **privado**. Crear una clase interna con un método que modifique el atributo de la clase externa y llame al método de la clase externa. En un segundo método de la clase externa, crear un objeto de la clase interna e invocar a su método, después mostrar el efecto en el objeto de la clase externa.

19. Repetir el Ejercicio 18 utilizando una clase interna anónima.

20. Crear una clase que contenga una clase interna **estática**. En el método **main( )**, crear una instancia de la clase interna.

21. Crear una **interfaz** que contenga una clase interna **estática**. Implementar esta **interfaz** y crear una instancia de la clase interna.

22. Crear una clase que contenga una clase interna que contenga a su vez otra clase interna. Repetir lo mismo usando clases internas **estática**. Fijarse en los nombres de los archivos **.class** producidos por el compilador.

23. Crear una clase con una clase interna. En una clase separada, hacer una instancia de la clase interna.

24. Crear una clase con una clase interna que tiene un constructor distinto del constructor por defecto. Crear una segunda clase con una clase interna que hereda de la primera clase interna.

25. Reparar el problema de **ErrorViento.java**.

26. Modificar **Secuencia.java** añadiendo un método **obtenerRSelector( )** que produce una implementación diferente de la **interfaz Selector** que se mueve hacia atrás de la secuencia desde el final al principio.

27. Crear una **interfaz U** con tres métodos. Crear una clase **A** con un método que produce una referencia a **U** construyendo una clase interna anónima. Crear una segunda clase **B** que contenga un array de **U**. **B** debería tener un método que acepte y almacene una referencia a **U** en el array, un segundo método que establece una referencia dentro del array (especificada por el parámetro del método) a **null,** y un tercer método que se mueve a través del array e invoca a los métodos de **U**. En el método **main( )**, crear un grupo de objetos **A** y un único **B**. Rellenar el **B** con referencias **U** producidas por los objetos **A**. Utilizar el **B** para invocar de nuevo a todos los objetos **A**. Eliminar algunas de las referencias **U** de **B**.

28. En **ControlesInvernadero.java**, añadir clases internas **Evento** que enciendan y apaguen ventiladores.

29. Mostrar que una clase interna tiene acceso a los elementos **privados** de su clase externa. Determinar si también se cumple a la inversa.

9: Guardar objetos

Es un programa bastante simple que sólo tiene una cantidad fija de objetos cuyos periodos de vida son conocidos.

En general, los programas siempre estarán creando nuevos objetos, en base a algún criterio que sólo se conocerá en tiempo de ejecución. No se sabrá hasta ese momento la cantidad o incluso el tipo exacto de objetos necesarios. Para solucionar el problema de programación general, hay que crear cualquier número de objetos, en cualquier momento, en cualquier sitio. Por tanto, no se puede confiar en crear una referencia con nombre que guarde cada objeto:

```
MiObjeto miReferencia;
```

dado que, de hecho, nunca se sabrá cuántas se necesitarán.

Para solucionar este problema tan esencial, Java tiene distintas maneras de guardar los objetos (o mejor, referencias a objetos). El tipo incrustado es el array, lo cual ya hemos discutido anteriormente. Además, la biblioteca de utilidades de Java tiene un conjunto razonablemente completo de *clases contenedoras* (conocidas también como *clases colección*, pero dado que las bibliotecas de Java 2 usan el nombre **Collection** para hacer referencia a un subconjunto particular de la biblioteca, usaremos el término más genérico "contenedor"). Los contenedores proporcionan formas sofisticadas de guardar e incluso manipular objetos.

Arrays

La mayoría de la introducción necesaria a los arrays se encuentra en la última sección del Capítulo 4, que mostraba cómo definir e inicializar un array. El propósito de este capítulo es el almacenamiento de objetos, y un array es justo una manera de guardar objetos. Pero hay muchas otras formas de guardar objetos, así que, ¿qué hace que un array sea tan especial?

Hay dos aspectos que distinguen a los arrays de otros tipos de contenedores: la eficiencia y el tipo. El array es la forma más eficiente que proporciona Java para almacenar y acceder al azar a una secuencia de objetos (verdaderamente, referencias a objeto). El array es una secuencia lineal simple, que hace rápidos los accesos a elementos, pero se paga por esta velocidad: cuando se crea un objeto array, su tamaño es limitado y no puede variarse durante la vida de ese objeto array. Se podría sugerir crear un array de un tamaño particular y, después, si se acaba el espacio, crear uno nuevo y mover todas las referencias del viejo al nuevo. Éste es el comportamiento de la clase **ArrayList**, que será estudiada más adelante en este capítulo. Sin embargo, debido a la sobrecarga de esta flexibilidad de tamaño, un **ArrayList** es mucho menos eficiente que un array.

La clase contenedora **Vector** de C++ *no* conoce el tipo de objetos que guarda, pero tiene un inconveniente diferente cuando se compara con los arrays de Java: el **operador[]** de **vector** de C++ no hace comprobación de límites, por lo que se puede ir más allá de su final[1]. En Java, hay comprobación de límites independientemente de si se está usando un array o un contenedor —se obtendrá una **excepción en tiempo de ejecución** si se exceden los límites. Como se aprenderá en el Capítulo 10, este tipo de excepción indica un error del programador, y por tanto, no es necesario comprobarlo en el código. Aparte de esto, la razón por la que el **vector** de C++ no comprueba los límites en todos los accesos es la velocidad —en Java se tiene sobrecarga de rendimiento constante de comprobación de límites todo el tiempo, tanto para arrays, como para contenedores.

Las otras clases contenedoras genéricas **List**, **Set** y **Map** que se estudiarán más adelante en este capítulo, manipulan los objetos como si no tuvieran tipo específico. Es decir, las tratan como de tipo **Object**, la clase raíz de todas las clases de Java. Esto trabaja bien desde un punto de vista: es necesario construir sólo un contenedor, y cualquier objeto Java entrará en ese contenedor. (Excepto por los tipos primitivos —que pueden ser ubicados en contenedores como constantes, utilizando las clases envolturas primitivas de Java, o como valores modificables envolviendo la propia clase.) Éste es el segundo lugar en el que un array es superior a los contenedores genéricos: cuando se crea un array, se crea para guardar un tipo específico. Esto significa que se da una comprobación de tipos en tiempo de compilación para evitar introducir el tipo erróneo o confundir el tipo que se espera. Por supuesto, Java evitará que se envíe el mensaje inapropiado a un objeto, bien en tiempo de compilación bien en tiempo de ejecución. Por tanto, no supone un riesgo mayor de una o de otra forma, sino que es simplemente más elegante que sea el compilador el que señale el error, además de ser más rápido en tiempo de ejecución, y así habrá menos probabilidades de sorprender al usuario final con una excepción.

En aras de la eficiencia y de la comprobación de tipos, siempre merece la pena intentar usar un array si se puede. Sin embargo, cuando se intenta solucionar un problema más genérico, los arrays pueden ser demasiado restrictivos. Después de ver los arrays, el resto de este capítulo se dedicará a las clases contenedoras proporcionadas por Java.

Los arrays son objetos de primera clase

Independientemente del tipo de array con el que se esté trabajando, el identificador de array es, de hecho, una referencia a un objeto verdadero que se crea en el montículo. Éste es el objeto que mantiene las referencias a los otros objetos, y puede crearse implícitamente como parte de la sintaxis de inicialización del atributo, o explícitamente mediante una sentencia **new**. Parte del objeto array (de hecho, el único atributo o método al que se puede acceder) es el miembro **length** de sólo lectura que dice cuántos elementos pueden almacenarse en ese array objeto. La sintaxis '**[]**' es el otro acceso que se tiene al array objeto.

[1] Es posible, sin embargo, preguntar por el tamaño del **vector**, y el método **at( )** *sí* que hará comprobación de límites.

El ejemplo siguiente muestra las distintas formas de inicializar un array, y cómo se pueden asignar las referencias array a distintos objetos array. También muestra que los arrays de objetos y los arrays de tipos primitivos son casi idénticos en uso. La única diferencia es que los arrays de objetos almacenan referencias, mientras que los arrays de primitivas guardan los valores primitivos directamente.

```
//: c09:TerminoArray.java
// Inicialización y reasignación de arrays.

class Mitologia {} // Una pequeña criatura mítica

public class TerminoArray {
  public static void main(String[] args) {
    // Arrays de objetos:
    Mitologia[] a; // Referencia Null
    Mitologia[] b = new Mitologia[5]; // Referencias Null
    Mitologia[] c = new Mitologia[4];
    for(int i = 0; i < c.length; i++)
      c[i] = new Mitologia();
    // Inicialización de agregados:
    Mitologia[] d = {
      new Mitologia(), new Mitologia(), new Mitologia()
    };
    // Inicialización dinámica de agregados:
    a = new Mitologia[] {
      new Mitologia(), new Mitologia()
    };
    System.out.println("a.length=" + a.length);
    System.out.println("b.length = " + b.length);
    // Las referencias internas del array se
    // inicializan automáticamente a null:
    for(int i = 0; i < b.length; i++)
      System.out.println("b[" + i + "]=" + b[i]);
    System.out.println("c.length = " + c.length);
    System.out.println("d.length = " + d.length);
    a = d;
    System.out.println("a.length = " + a.length);

    // Arrays de datos primitivos:
    int[] e; // Referencia null
    int[] f = new int[5];
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
```

```
    // Error de compilación: variable e sin inicializar:
    //!System.out.println("e.length=" + e.length);
    System.out.println("f.length = " + f.length);
    // Los datos primitivos de dentro del array se
    // inicializan automáticamente a cero:
    for(int i = 0; i < f.length; i++)
      System.out.println("f[" + i + "]=" + f[i]);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    e = new int[] { 1, 2 };
    System.out.println("e.length = " + e.length);
  }
} ///:~
```

He aquí la salida del programa:

```
b.length = 5
b[0]=null
b[1]=null
b[2]=null
b[3]=null
b[4]=null
c.length = 4
d.length = 3
a.length = 3
a.length = 2
f.length = 5
f[0]=0
f[1]=0
f[2]=0
f[3]=0
f[4]=0
g.length = 4
h.length = 3
e.length = 3
e.length = 2
```

Inicialmente una simple referencia **null** (traducción errónea), y el compilador evita que se haga nada con ella hasta que se haya inicializado adecuadamente. El array **b** se inicializa a un array de referencias **Mitología**, pero, de hecho, no se colocan objetos **Mitología** en ese array. Sin embargo, se sigue pudiendo preguntar por el tamaño del array, dado que **b** apuntó a un objeto legítimo. Esto presenta un pequeño inconveniente: no se puede averiguar cuántos elementos hay *en* el array, puesto que **length** dice sólo cuántos elementos *se pueden* ubicar en el array; es decir, el tamaño del objeto array, no el número de objetos que alberga. Sin embargo, cuando se crea un objeto array sus

referencias se inicializan automáticamente a **null**, por lo que se puede ver si una posición concreta del array tiene un objeto, comprobando si es o no **null**. De forma análoga, un array de tipos primitivos se inicializa automáticamente a cero en el caso de los números, **(char)0** en el caso de **caracteres**, y **false** si se trata de **lógicos**.

El array **c** muestra la creación del objeto array seguida de la asignación de objetos **Mitología** a las posiciones del mismo. El array **d** muestra la sintaxis de "inicialización de agregados" que hace que se cree el objeto array (implícitamente en el montículo, con **new**, al igual que ocurre con el array **c**) *y* que se inicialice con objetos **Mitología**, todo ello en una sentencia.

La siguiente inicialización del array podría definirse como "inicialización dinámica de agregados". La inicialización de agregados usada por **d** debe usarse al definir **d**, pero con la segunda sintaxis se puede crear e inicializar un objeto array en cualquier lugar. Por ejemplo, supóngase que **esconder( )** sea un método que toma un array de objetos **Mitología**. Se podría invocar diciendo:

```
esconder(d);
```

pero también se puede crear dinámicamente el array al que se desea pasar el parámetro:

```
esconder(new Mitologia[] ) { new Mitologia (), new Mitologia() });
```

En algunas situaciones, esta nueva sintaxis proporciona una forma más adecuada de escribir código.

La expresión

```
a = d;
```

muestra cómo se puede tomar una referencia adjuntada a un objeto array y asignársela a otro objeto array, exactamente igual que se puede hacer con cualquier otro tipo de referencia a objeto. Ahora tanto **a** como **b** apuntan al mismo objeto array del montículo.

La segunda parte de **TamanioArray.java** muestra que los arrays de tipos primitivos funcionan exactamente igual que los arrays de objetos *excepto* que los arrays de tipos primitivos guardan los valores directamente.

Contendores de datos primitivos

Las clases contenedoras sólo pueden almacenar referencias a objetos. Sin embargo, se puede crear un array para albergar directamente tipos primitivos, al igual que referencias a objetos. *Es* posible utilizar las clases envoltorio ("wrapper") como **Integer**, **Double**, etc. para ubicar valores primitivos dentro de un contenedor, pero estas clases pueden ser tediosas de usar. Además, es mucho más eficiente crear y acceder a un array de datos primitivos que a un contenedor de objetos envoltorio.

Por supuesto, si se está usando un tipo primitivo y se necesita la flexibilidad de un contenedor que se expanda automáticamente cuando se necesita más espacio, el array no será suficiente, por lo que uno se verá obligado a usar un contenedor de objetos envoltorio. Se podría pensar que debería haber un tipo especializado de **ArrayList** por cada tipo de dato primitivo, pero Java no proporciona

esto. Quizás algún día cualquier tipo de mecanismo plantilla proporcione un método que haga que Java maneje mejor este problema[2].

Devolver un array

Suponga que se está escribiendo un método y no se quiere que devuelva sólo un elemento, sino un conjunto de elementos. Los lenguajes como C y C++ hacen que esto sea difícil porque no se puede devolver un array sin más, hay que devolver un puntero a un array. Esto supone problemas pues se vuelve complicado intentar controlar la vida del array, lo que casi siempre acaba llevando a problemas de memoria.

Java sigue un enfoque similar, pero simplemente "devuelve un array". Por supuesto que, de hecho, se está devolviendo una referencia a un array, pero con Java uno nunca tiene por qué preocuparse de ese array —estará por ahí mientras se necesite, y el recolector de basura no lo eliminará hasta que deje de utilizarse.

Como ejemplo, considere que se devuelve un array de **cadenas de caracteres**:

```
//: c09:Helado.java
// Métodos que devuelven arrays.

public class Helado {
  static String[] sabor = {
    "Chocolate", "Fresa",
    "Vainilla", "Menta",
    "Moca y almendras", "Ron con pasas",
    "Praline", "Turrón"
  };
  static String[] ConjuntoSabores(int n) {
    // Forzar a que sea positivo y dentro de los límites:
    n = Math.abs(n) % (sabor.length + 1);
    String[] resultados = new String[n];
    boolean[] seleccionado =
      new boolean[sabor.length];
    for (int i = 0; i < n; i++) {
      int t;
      do
        t = (int)(Math.random() * sabor.length);
      while (seleccionado[t]);
      resultados[i] = sabor[t];
      seleccionado[t] = true;
    }
```

[2] Éste es uno de los puntos en los que C++ es enormemente superior a Java, dado que C++ soporta los *tipos parametrizados* haciendo uso de la palabra clave **template**.

```
      return resultados;
   }
   public static void main(String[] args) {
      for(int i = 0; i < 20; i++) {
         System.out.println(
            "ConjuntoSabores(" + i + ") = ");
         String[] fl = ConjuntoSabores(sabor.length);
         for(int j = 0; j < fl.length; j++)
            System.out.println("\t" + fl[j]);
      }
   }
} ///:~
```

El método **ConjuntoSabores( )** crea un array de **cadenas de caracteres** llamado **resultados**. El tamaño del array es **n**, determinado por el parámetro que se le pasa al método. Posteriormente, procede a elegir sabores de manera aleatoria a partir del array **sabores** y a ubicarlos en **resultados**, que es lo que finalmente devuelve. Devolver el array es exactamente igual que devolver cualquier otro objeto —es una referencia. No es importante en este momento el que el array se haya creado dentro de **ConjuntoSabores( )**, o que el array se haya creado en cualquier otro sitio. El recolector de basura se encarga de limpiar el array una vez que se ha acabado con él, pero mientras tanto, éste seguirá vivo.

Aparte de lo ya comentado, nótese que **ConjuntoSabores( )** elige sabores al azar, asegurando para cada una de las elecciones que ésta no ha salido antes. Esto se hace en un bucle **do** que se encarga de hacer selecciones al azar hasta encontrar una que ya no está en el array **seleccionado**. (Por supuesto, también se podría realizar una comparación de **cadenas de caracteres** para ver si la selección hecha al azar ya estaba en el array **resultados**, pero las comparaciones de **cadenas de caracteres** son ineficientes.) Si tiene éxito, añade la entrada y pasa al siguiente (se incrementa **i**).

El método **main( )** imprime 20 conjuntos completos de sabores, por lo que se puede ver que **ConjuntoSabores( )** elige los sabores en orden aleatorio cada vez. Esto se ve mejor si se redirecciona la salida a un archivo. Y al recorrer el archivo, recuérdese que uno simplemente *quiere* el helado, no lo *necesita*.

La clase **Arrays**

En **java.util** se encuentra la clase **Arrays**, capaz de mantener un conjunto de métodos **estáticos** que llevan a cabo funciones de utilidad para arrays. Tiene cuatro funciones básicas: **equals( )** para comparar la igualdad de dos arrays; **fill( )** para rellenar un array con un valor; **sort( )** para ordenar el array; y **binarySearch( )** para encontrar un dato en un array ordenado. Todos estos métodos están sobrecargados para todos los tipos de datos primitivos y **objetos**. Además, hay un método simple **asList( )** que hace que un array se convierta en un contendor **List**, del cual se aprenderá más adelante en este capítulo.

A la vez que útil, la clase **Arrays** puede dejar de ser completamente funcional. Por ejemplo, sería bueno ser capaces de imprimir los elementos de un array sin tener que codificar el código **for** a mano cada vez. Como se verá, el método **fill( )** sólo toma un único valor y lo posiciona en el array, por lo que si se deseaba —por ejemplo— rellenar un array con números generados al azar, **fill( )** no es suficiente.

Por consiguiente, tiene sentido complementar la clase **Arrays** con alguna utilidad adicional, que se ubicará por comodidad en el paquete **com.bruceeckel.util**. Estas utilidades permiten imprimir un array de cualquier tipo, y rellenan un array con valores u objetos creados por un objeto denominado *generador* que cada uno puede definir.

Dado que es necesario crear código para cada tipo primitivo al igual que para **Object**, hay muchísimo código prácticamente duplicado[3].Por ejemplo, se requiere una interfaz "generador" por cada tipo, puesto que el valor de retorno de **siguiente( )** debe ser distinto en cada caso:

```
//: com:bruceeckel:util:Generador.java
package com.bruceeckel.util;
public interface Generador {
  Object siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorBoolean.java
package com.bruceeckel.util;
public interface GeneradorBoolean {
  boolean siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorByte.java
package com.bruceeckel.util;
public interface GeneradorByte {
  byte siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorChar.java
package com.bruceeckel.util;
public interface GeneradorChar {
  char siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorShort.java
package com.bruceeckel.util;
public interface GeneradorShort {
  short siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorInt.java
package com.bruceeckel.util;
public interface GeneradorInt {
  int siguiente();
} ///:~

//: com:bruceeckel:util:GeneradorLong.java
```

[3] El programador de C++ notará cuánto código podría colapsarse con la utilizacón de parámetros por defecto y plantillas. El programador de Python notará que esta biblioteca sería completamente innecesaria en este último lenguaje.

```
package com.bruceeckel.util;
public interface GeneradorLong {
  long siguiente();
} ///:~
```

```
//: com:bruceeckel:util:GeneradorFloat.java
package com.bruceeckel.util;
public interface GeneradorFloat {
  float siguiente();
} ///:~
```

```
//: com:bruceeckel:util:GeneradorDouble.java
package com.bruceeckel.util;
public interface GeneradorDouble {
  double siguiente();
} ///:~
```

Arrays2 contiene varias funciones **escribir( )**, sobrecargadas para cada tipo. Se puede simplemente imprimir un array, de forma que se pueda añadir un mensaje antes de que se imprima, o se puede imprimir un rango de elementos dentro de un array. El código de añadir método **escribir( )** es casi autoexplicatorio:

```
//: com:bruceeckel:util:Arrays2.java
// Un suplemento para java.util.Arrays, que proporciona
// funcionalidad adicional útil para trabajar
// con arrays. Permite imprimir un array,
// que puede ser rellenado a través un objeto "generador"
// definido por el usuario.
package com.bruceeckel.util;
import java.util.*;

public class Arrays2 {
  private static void
  start(int de, int para, int longitud) {
    if(de != 0 || para != longitud)
      System.out.print("["+ de +":"+ para +"] ");
    System.out.print("(");
  }
  private static void fin() {
    System.out.println(")");
  }
  public static void escribir(Object[] a) {
    escribir(a, 0, a.length);
  }
  public static void
  print(String mensaje, Object[] a) {
    System.out.escribir(mensaje + " ");
```

```
    escribir(a, 0, a.length);
  }
  public static void
  escribir(Object[] a, int de, int para){
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para -1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(boolean[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, boolean[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(boolean[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para -1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(byte[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, byte[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(byte[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para -1)
        System.out.print(", ");
```

```
    }
    fin();
  }
  public static void escribir(char[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, char[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(char[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para -1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(short[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, short[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(short[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(int[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, int[] a) {
    System.out.print(mensajes + " ");
```

```
    escribir(a, 0, a.length);
  }
  public static void
  escribir(int[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(long[] a) {
    escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, long[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(long[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
    }
    fin();
  }
  public static void escribir(float[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, float[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(float[] a, int de, int para) {
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
```

```
    }
    fin();
  }
  public static void escribir(double[] a) {
      escribir(a, 0, a.length);
  }
  public static void
  escribir(String mensaje, double[] a) {
    System.out.print(mensaje + " ");
    escribir(a, 0, a.length);
  }
  public static void
  escribir(double[] a, int de, int para){
    comenzar(de, para, a.length);
    for(int i = de; i < para; i++) {
      System.out.print(a[i]);
      if(i < para - 1)
        System.out.print(", ");
    }
    fin();
  }
  // Rellenar un array utilizando un generador:
  public static void
  rellenar(Object[] a, Generador gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(Object[] a, int de, int para,
        Generador gen){
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(boolean[] a, GeneradorBoolean gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(boolean[] a, int de, int para,
      GeneradorBoolean gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(byte[] a, GeneradorByte gen) {
      rellenarl(a, 0, a.length, gen);
```

```
  }
  public static void
  rellenar(byte[] a, int de, int para,
      GeneradorByte gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(char[] a, GeneradorChar gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(char[] a, int de, int para,
      GeneradorChar gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(short[] a, GeneradorShort gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(short[] a, int de, int para,
      GeneradorShort gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(int[] a, GeneradorInt gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(int[] a, int de, int para,
      GeneradorInt gen) {
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(long[] a, GeneratorLong gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(long[] a, int de, int para,
      GeneradorLong gen) {
    for(int i = de; i < para; i++)
```

```
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(float[] a, GeneradorFloat gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(float[] a, int de, int para,
       GeneradorFloat gen) {
    for(int i = from; i < para; i++)
      a[i] = gen.siguiente();
  }
  public static void
  rellenar(double[] a, GeneradorDoble gen) {
      rellenar(a, 0, a.length, gen);
  }
  public static void
  rellenar(double[] a, int de, int para,
       DoubleGenerator gen){
    for(int i = de; i < para; i++)
      a[i] = gen.siguiente();
  }
  private static Random r = new Random();
  public static class GeneradorBooleanAleatorio
  implements GeneradorBoolean {
    public boolean siguiente() {
      return r.nextBoolean();
    }
  }
  public static class GeneradorByteAleatorio
  implements GeneradorByte {
    public byte siguiente() {
      return (byte)r.nextInt();
    }
  }
  static String fuente =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
    "abcdefghijklmnopqrstuvwxyz";
  static char[] fuenteArray = fuente.toCharArray();
  public static class GeneradorCharAleatorio
  implement GeneradorChar {
    public char siguiente() {
      int pos = Math.abs(r.nextInt());
      return fuenteArray[pos % fuenteArray.length];
    }
```

```
  }
  public static class GeneradorStringAleatorio
  implements Generador {
    private int lon;
    private GeneradorCharAleatorio cg =
      new GeneradorCharAleatorio();
    public GeneradorStringAleatorio(int longitud) {
      lon = longitud;
    }
    public Object siguiente() {
      char[] buf = new char[lon];
      for(int i = 0; i < lon; i++)
        buf[i] = cg.siguiente();
      return new String(buf);
    }
  }
  public static class GeneradorShorAleatorio
  implements GeneradorInt {
    public short siguiente() {
      return (short)r.nextInt();
    }
  }
  public static class GeneradorIntAleatorio
  implements GeneratorInt {
    private int mod = 10000;
    public GeneradorIntAleatorio() {}
    public GeneradorIntAleatorio(int modulo) {
      mod = modulo;
    }
    public int siguiente() {
      return r.nextInt() % mod;
    }
  }
  public static class GeneradorLongAleatorio
  implement GeneradorLong {
    public long siguiente() { return r.nextLong(); }
  }
  public static class GeneradorFloatAleatorio
  implements GeneradorFloat {
    public float siguiente() { return r.nextFloat(); }
  }
  public static class GeneradorDoubleAleatorio
  implements GeneradorDouble {
    public double siguiente() {return r.nextDouble();}
  }
} ///:~
```

Para rellenar un array utilizando un generador, el método **rellenar()** toma una referencia a una **interfaz** generadora adecuada, que tiene un método **siguiente()** que de alguna forma producirá un objeto del tipo correcto (dependiendo de cómo se implemente la interfaz). El método **rellenar()** simplemente invoca a **siguiente()** hasta que se ha rellenado el rango deseado. Ahora, se puede crear cualquier generador implementando la **interfaz** adecuada, y luego utilizar el generador con el método **rellenar()**.

Los generadores de datos aleatorios son útiles para hacer pruebas, por lo que se crea un conjunto de clases internas para implementar las interfaces generadoras de datos primitivos, al igual que el generador de cadenas de caracteres **String** para representar a un **Objeto**. Se puede ver que **GeneradorStringAleatorio** usa **GeneradorCharAleatorio** para rellenar un array de caracteres, que después se convierte en una cadena de caracteres (**String**). El tamaño del array viene determinado por el parámetro pasado al constructor.

Para generar números que no sean demasiado grandes, **GeneradorIntAleatorio** toma por defecto un módulo de 10.000, pero el constructor sobrecargado permite seleccionar un valor menor.

He aquí un programa para probar la biblioteca y demostrar cómo se usa:

```
//: c09:PruebaArrays2.java
// Probar y demostrar las utilidades de Arrays2
import com.bruceeckel.util.*;

public class PruebaArrays2 {
  public static void main(String[] args) {
    int tamanio = 6;
    // O tomar el tamaño de la lista de parámetros:
    if(args.length != 0)
      tamanio= Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[tamanio];
    byte[] a2 = new byte[tamanio];
    char[] a3 = new char[tamanio];
    short[] a4 = new short[tamanio];
    int[] a5 = new int[tamanio];
    long[] a6 = new long[tamanio];
    float[] a7 = new float[tamanio];
    double[] a8 = new double[tamanio];
    String[] a9 = new String[tamanio];
    Arrays2.rellenar(a1,
      new Arrays2.GeneradorBooleanAleatorio());
    Arrays2.escribir(a1);
    Arrays2.escribir("a1 = ", a1);
    Arrays2.escribir(a1, tamanio/3, tamanio/3 + tamanio/3);
    Arrays2.rellenar(a2,
      new Arrays2.GeneradorByteAleatorio());
    Arrays2.escribir(a2);
    Arrays2.escribir("a2 = ", a2);
    Arrays2.escribir(a2, tamanio/3, tamanio/3 + tamanio/3);
```

```
      Arrays2.rellenar(a3,
        new Arrays2.GeneradorCharAleatorio());
      Arrays2.escribir(a3);
      Arrays2.escribir("a3 = ", a3);
      Arrays2.escribir(a3, tamanio/3, tamanio/3 + tamanio/3);
      Arrays2.rellenar(a4,
        new Arrays2.GeneradorShortAleatorio());
      Arrays2.escribir(a4);
      Arrays2.escribir("a4 = ", a4);
      Arrays2.escribir(a4, tamanio/3, tamanio/3 + tamanio/3);
      Arrays2.rellenar(a5,
        new Arrays2.GeneradorIntAleatorio());
      Arrays2.escribir(a5);
      Arrays2.escribir("a5 = ", a5);
      Arrays2.escribir(a5, tamanio/3, tamanio/3 + tamanio/3);
      Arrays2.rellenar(a6,
        new Arrays2.GeneradorLongAleatorio());
      Arrays2.escribir(a6);
      Arrays2.escribir("a6 = ", a6);
      Arrays2.escribir(a6, tamanio/3,tamanio/3 + tamanio/3);
      Arrays2.rellenar(a7,
        new Arrays2.GeneradorFloatAleatorio());
      Arrays2.escribir(a7);
      Arrays2.escribir("a7 = ", a7);
      Arrays2.escribir(a7, tamanio/3, tamanio/3 + tamanio/3);
      Arrays2.rellenar(a8,
        new Arrays2.GeneradorDoubleAleatorio());
      Arrays2.escribir(a8);
      Arrays2.escribir("a8 = ", a8);
      Arrays2.escribir(a8, tamanio/3, tamanio/3 + tamanio/3);
      Arrays2.rellenar(a9,
        new Arrays2.GeneradorStringAleatorio(7));
      Arrays2.escribir(a9);
      Arrays2.escribir("a9 = ", a9);
      Arrays2.escribir(a9, tamanio/3, tamanio/3 + tamanio/3);
   }
} ///:~
```

El parámetro **tamanio** tiene un valor por defecto, pero también se puede establecer a partir de la línea de comandos.

Rellenar un array

La biblioteca estándar de Java **Arrays** también tiene un método **rellenar (fill( ))**, pero es bastante trivial —sólo duplica un único valor en cada posición, o en el caso de los objetos, copia la misma re-

ferencia a todas las posiciones. Utilizando **Arrays2.escribir( )**, se pueden demostrar fácilmente los métodos **Arrays.fill( )**:

```
//: c09:RellenarArrays.java
// Usando Arrays.fill()
import com.bruceeckel.util.*;
import java.util.*;

public class RellenarArrays {
  public static void main(String[] args) {
    int tamanio = 6;
    // O tomar el tamaño de la línea de comandos:
    if(args.length != 0)
      tamanio = Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[tamanio];
    byte[] a2 = new byte[tamanio];
    char[] a3 = new char[tamanio];
    short[] a4 = new short[tamanio];
    int[] a5 = new int[tamanio];
    long[] a6 = new long[tamanio];
    float[] a7 = new float[tamanio];
    double[] a8 = new double[tamanio];
    String[] a9 = new String[tamanio];
    Arrays.fill(a1, true);
    Arrays2.escribir("a1 = ", a1);
    Arrays.fill(a2, (byte)11);
    Arrays2.escribir("a2 = ", a2);
    Arrays.fill(a3, 'x');
    Arrays2.escribir("a3 = ", a3);
    Arrays.fill(a4, (short)17);
    Arrays2.escribir("a4 = ", a4);
    Arrays.fill(a5, 19);
    Arrays2.escribir("a5 = ", a5);
    Arrays.fill(a6, 23);
    Arrays2.escribir("a6 = ", a6);
    Arrays.fill(a7, 29);
    Arrays2.escribir("a7 = ", a7);
    Arrays.fill(a8, 47);
    Arrays2.escribir("a8 = ", a8);
    Arrays.fill(a9, "Hola");
    Arrays2.escribir("a9 = ", a9);
    // Manipular rangos:
    Arrays.fill(a9, 3, 5, "Mundo");
    Arrays2.escribir("a9 = ", a9);
  }
} ///:~
```

Se puede o bien rellenar todo el array, o —como se ve en las dos últimas sentencias —un rango de elementos. Pero dado que sólo se puede proporcionar un valor a usar en el relleno si se usa **Arrays.fill( )**, los métodos **Arrays2.rellenar( )** producen resultados mucho más interesantes.

Copiar un array

La biblioteca estándar de Java proporciona un método **estático**, llamado **System.arraycopy( )**, que puede hacer copias mucho más rápidas de arrays que si se usa un bucle **for** para hacer la copia a mano. **System.arraycopy( )** está sobrecargado para manejar todos los tipos. He aquí un ejemplo que manipula un array de **enteros:**

```
//: c09:CopiarArrays.java
// Usando System.arraycopy()
import com.bruceeckel.util.*;
import java.util.*;

public class CopiarArrays {
  public static void main(String[] args) {
    int[] i = new int[25];
    int[] j = new int[25];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    Arrays2.escribir("i = ", i);
    Arrays2.escribir("j = ", j);
    System.arraycopy(i, 0, j, 0, i.length);
    Arrays2.escribir("j = ", j);
    int[] k = new int[10];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    Arrays2.escribir("k = ", k);
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    Arrays2.print("i = ", i);
    // Objetos:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    Arrays2.escribir("u = ", u);
    Arrays2.escribir("v = ", v);
    System.arraycopy(v, 0,
      u, u.length/2, v.length);
    Arrays2.escribir("u = ", u);
  }
} ///:~
```

Los parámetros de **arraycopy( )** son el array fuente, el desplazamiento del array fuente a partir del cual comenzar la copia, el array de destino, el desplazamiento dentro del array de destino en el que comenzar a copiar, y el número de elementos a copiar. Naturalmente, cualquier violación de los límites del array causaría una excepción.

El ejemplo muestra que pueden copiarse, tanto los arrays de datos primitivos, como los de objetos. Sin embargo, si se copia un array de objetos, solamente se copian las referencias —no hay duplicación de los objetos en sí. A esto se le llama *copia superficial.* (Véase Apéndice A.)

Comparar arrays

La clase **Arrays** proporciona un método sobrecargado **equals( )** para comparar arrays enteros y ver si son iguales. Otra vez, se trata de un método sobrecargado para todos los tipos de datos primitivos y para **Objetos**. Para que dos arrays sean iguales, deben tener el mismo número de elementos y además cada elemento debe ser equivalente a su elemento correspondiente en el otro array, utilizando el método **equals( )** para cada elemento. (En el caso de datos primitivos, se usa la clase de su envoltorio **equals( )**; por ejemplo, se usa **Integer.equals( )** para **int**.) He aquí un ejemplo:

```
//: c09:CompararArrays.java
// Usando Arrays.equals()
import java.util.*;

public class CompararArrays {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = new int[10];
    Arrays.fill(a1, 47);
    Arrays.fill(a2, 47);
    System.out.println(Arrays.equals(a1, a2));
    a2[3] = 11;
    System.out.println(Arrays.equals(a1, a2));
    String[] s1 = new String[5];
    Arrays.fill(s1, "Hi");
    String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
    System.out.println(Arrays.equals(s1, s2));
  }
} ///:~
```

Originalmente, **a1** y **a2** son exactamente iguales, de forma que la salida es "verdadero", pero al cambiar uno de los elementos, la segunda línea de la salida será "falso". En este último caso, todos los elementos de **s1** apuntan al mismo objeto, pero **s2** tiene cinco únicos objetos. Sin embargo, la igualdad de los arrays se basa en los contenidos (mediante **Object.equals( )**) por lo que el resultado es "verdadero".

Comparaciones de elementos de arrays

Una de las características que faltan en las bibliotecas de Java 1.0 y 1.1 son las operaciones logarítmicas —incluso la ordenación simple. Ésta era una situación bastante confusa para alguien que esperara una biblioteca estándar adecuada. Afortunadamente, Java 2 remedia esta situación, al menos para el problema de ordenación.

El problema de escribir código de ordenación genérico consiste en que esa ordenación debe llevar a cabo comparaciones basadas en el tipo del objeto. Por supuesto, un enfoque es escribir un método de ordenación distinto para cada tipo distinto, pero habría que ser capaz de reconocer que esto no produce código fácilmente reutilizable para tipos nuevos.

Un primer objetivo del diseño de programación es "separar los elementos que cambian de los que permanecen igual", y aquí el código que sigue igual es el algoritmo general de ordenación, pero lo que cambia de un uso al siguiente es la forma de comparar los objetos. Por tanto, en vez de incluir el código de comparación en muchas rutinas de ordenación se usa la técnica de *retrollamadas.* Con una llamada hacia atrás, la parte de código que varía de caso a caso está encapsulada dentro de su propia clase, y la parte de código que es siempre la misma puede invocar hacia atrás al código que cambia. De esa forma, se pueden hacer objetos diferentes para expresar distintas formas de comparación y alimentar con ellos el mismo código de ordenación.

En Java 2 hay dos formas de proporcionar funcionalidad de comparación. La primera es con el *método de comparación natural*, que se comunica con una clase implementando la interfaz **java.lang.Comparable**. Se trata de una interfaz bastante simple con un único método **compareTo( )**. Este método toma otro **Objeto** como parámetro, y produce un valor negativo si el parámetro es menor que el objeto actual, cero si el parámetro es igual, y un valor positivo si el parámetro es mayor que el objeto actual.

He aquí una clase que implementa **Comparable** y demuestra la comparación utilizando el método **Arrays.sort( )** de la biblioteca estándar de Java:

```
//: c09:TipoComp.java
// Implementando Comparable en una clase.
import com.bruceeckel.util.*;
import java.util.*;

public class TipoComp implements Comparable {
  int i;
  int j;
  public TipoComp(int n1, int n2) {
    i = n1;
    j - n2;
  }
  public String toString() {
    return "[i = " + i + ", j = " + j + "]";
  }
```

```
  public int compareTo(Object rv) {
    int rvi = ((TipoComp)rv).i;
    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
  }
  private static Random r = new Random();
  private static int intAleatorio() {
    return Math.abs(r.nextInt()) % 100;
  }
  public static Generador generador() {
    return new Generador() {
      public Object siguiente() {
        return new TipoComp(intAleatorio(),intAleatorio());
      }
    };
  }
  public static void main(String[] args) {
    TipoComp[] a = new TipoComp[10];
    Arrays2.rellenar(a, generador());
    Arrays2.escribir("antes de ordenar, a = ", a);
    Arrays.sort(a);
    Arrays2.escribir("despues de ordenar, a = ", a);
  }
} ///:~
```

Cuando se define la función de comparación, uno es responsable de decidir qué significa comparar un objeto con otro. Aquí, sólo se usan los valores **i** para la comparación, ignorando los valores **j**.

El método **estático intAleatorio( )** produce valores positivos entre 0 y 100, y el método **generador( )** produce un objeto que implementa la interfaz **Generador**, creando una clase interna anónima (ver Capítulo 8). Así se crean objetos **TipoComp** inicializándolos con valores al azar. En **main ( )** se usa el generador para rellenar un array de **TipoComp**, que se ordena después. Si no se hubiera implementado **Comparable**, se habría obtenido un mensaje de error de tiempo de compilación al tratar de invocar a **sort( )**.

Ahora, supóngase que alguien nos pasa una clase que no implementa **Comparable**, o nos pasan esta clase que *sí* que implementa **Comparable**, pero decide que no le gusta cómo funciona y preferiríamos una función de comparación distinta. Para lograrlo, se usa el segundo enfoque de comparación de objetos, creando una clase separada que implementa una interfaz denominada **Comparator**. Ésta tiene dos métodos, **compare( )** y **equals( )**. Sin embargo, no se tiene que implementar **equals( )** excepto por necesidades de rendimiento especiales, dado que cada vez que se crea la clase ésta se hereda implícitamente de **Object**, que tiene un **equals( )**. Por tanto, se puede usar el método por defecto **equals( )** que devuelve un **object** y satisfacer el contrato impuesto por la interfaz.

La clase **Collections** (que se estudiará más tarde) contiene un **Comparator** simple que invierte el orden de ordenación natural. Éste se puede aplicar de manera sencilla al **TipoComp**:

```
//: c09:Inverso.java
// El comparador Collecions.reverseOrder().
import com.bruceeckel.util.*;
import java.util.*;

public class Inverso {
  public static void main(String[] args) {
    TipoComp[] a = new TipoComp[10];
    Arrays2.rellenar(a, TipoComp.generador());
    Arrays2.escribir("antes de ordenar, a = ", a);
    Arrays.sort(a, Collections.reverseOrder());
    Arrays2.escribir("despues de ordenar, a = ", a);
  }
} ///:~
```

La llamada a **Collections.reverseOrder( )** produce la referencia al **Comparator**.

Como segundo ejemplo, el **Comparator** siguiente compara objetos **TipoComp** basados en sus valores **j** en vez de en sus valores **i**:

```
//: c09:PruebaComparador.java
// Implementando un Comparator para una clase.
import com.bruceeckel.util.*;
import java.util.*;

class ComprobadorTipoComp implements Comparator {
  public int compare(Object o1, Object o2) {
    int j1 = ((TipoComp)o1).j;
    int j2 = ((TipoComp)o2).j;
    return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
  }
}

public class PruebaComparador {
  public static void main(String[] args) {
    TipoComp[] a = new TipoComp[10];
    Arrays2.rellenar(a, TipoComp.generador());
    Arrays2.escribir("antes de ordenar, a = ", a);
    Arrays.sort(a, new ComparadorTipoComp());
    Arrays2.escribir("despues de ordenar, a = ", a);
  }
} ///:~
```

El método **compare( )** debe devolver un entero negativo, cero o un entero positivo si el primer parámetro es menor que, igual o mayor que el segundo, respectivamente.

Ordenar un array

Con los métodos de ordenación incluidos, se puede ordenar cualquier array de tipos primitivos, y un array de objetos que, o bien implemente **Comparable,** o bien tenga un **Comparator** asociado. Éste rellena un gran agujero en las bibliotecas Java —se crea o no, ¡en Java 1.0 y 1.1 no había soporte para ordenar cadenas de caracteres! He aquí un ejemplo que genera objetos **String** y los ordena:

```
//: c09:OrdenarStrings.java
// Ordenando un array de Strings.
import com.bruceeckel.util.*;
import java.util.*;

public class OrdenarStrings {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.rellenar(sa,
      new Arrays2.GeneradorStringAleatorio(5));
    Arrays2.escribir("Antes de ordenar: ", sa);
    Arrays.sort(sa);
    Arrays2.escribir("Despues de ordenar: ", sa);
  }
} ///:~
```

Algo que uno notará de la salida del algoritmo de ordenación de cadenas de caracteres es que es *lexicográfico*, por lo que coloca en primer lugar las palabras que empiezan con letras mayúsculas, seguidas de todas las palabras que empiezan con minúsculas. (Las guías telefónicas suelen ordenarse así.) También se podría desear agrupar todas las palabras juntas independientemente de si empiezan con mayúsculas o minúsculas, lo cual se puede hacer definiendo una clase **Comparator**, y por consiguiente, sobrecargando el comportamiento por defecto de **Comparable** para cadenas de caracteres. Para su reutilización, ésta se añadirá al paquete “util”:

```
//: com:bruceeckel:util:ComparadorAlfabetico.java
// Manteniendo juntas las letras mayúsculas y minúsculas.
package com.bruceeckel.util;
import java.util.*;

public class ComparadorAlfabetico
implements Comparador{
  public int compare(Object o1, Object o2) {
    String s1 = (String)o1;
    String s2 = (String)o2;
    return s1.toLowerCase().compareTo(
      s2.toLowerCase());
  }
} ///:~
```

Cada **String** se convierte a minúsculas antes de esta comparación. El método **compareTo( )** incluido en **String** proporciona la funcionalidad deseada.

He aquí una prueba usando **ComparadorAlfabetico:**

```
//: c09:OrdenaAlfabeticamente.java
// Mantiene juntas las letras mayúsculas y minúsculas
import com.bruceeckel.util.*;
import java.util.*;

public class OrdenaAlfabeticamente {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.rellenar(sa,
      new Arrays2.GeneradorStringAleatorio(5));
    Arrays2.escribir("Antes de ordenar: ", sa);
    Arrays.sort(sa, new ComparadorAlfabetico());
    Arrays2.escribir("Despues de ordenar: ", sa);
  }
} ///:~
```

El algoritmo de ordenación usado en la biblioteca estándar de Java se diseñó para ser óptimo para el tipo de datos en particular a ordenar —algoritmo de ordenación rápida (Quicksort) para tipos primitivos, y un método de ordenación por mezcla *(merge sort)* en el caso de objetos. Por tanto, no sería necesario preocuparse por el rendimiento a menos que alguna herramienta de optimización indicara que el proceso de ordenación constituye un cuello de botella.

Buscar en un array ordenado

Una vez ordenado el array, se puede llevar a cabo una búsqueda rápida de algún elemento dentro del mismo utilizando **Arrays.binarySearch( )**. Sin embargo, es muy importante que no se trate de hacer uso de **binarySearch( )** en un array sin ordenar; los resultados serían impredecibles. El ejemplo siguiente usa un **GeneradorIntAleatorio** para rellenar un array, y después produce valores a buscar en el mismo:

```
//: c09:BuscarEnArray.java
// Usando Arrays.binarySearch().
import com.bruceeckel.util.*;
import java.util.*;

public class BuscarEnArray {
  public static void main(String[] args) {
    int[] a = new int[100];
    Arrays2. GeneradorIntAleatorio gen =
      new Arrays2.GeneradorIntAleatorio (1000);
    Arrays2.rellenar(a, gen);
```

```
      Arrays.sort(a);
      Arrays2.escribir("Array ordenado: ", a);
      while(true) {
        int r = gen.siguiente();
        int posicion = Arrays.binarySearch(a, r);
        if(posicion >= 0) {
          System.out.println("Localizacion de " + r +
            " es " + posicion + ", a[" +
            posicion + "] = " + a[posicion]);
          break; // sale del bucle while
        }
      }
    }
} ///:~
```

En el bucle **while** se generan valores aleatorios como datos a buscar, hasta que se encuentre uno de ellos.

Arrays.binarySearch( ) produce un valor mayor o igual a cero si se encuentra el elemento. Sino, produce un valor negativo que representa el lugar en el que debería insertarse el elemento si se estuviera manteniendo el array ordenado a mano. El valor producido es:

```
-(punto de inserción) - 1
```

El punto de inserción es el índice del primer elemento mayor que la clave, o **a.size( )**, si todos los elementos del array son menores que la clave especificada.

Si el array contiene elementos duplicados, no hay garantía de cuál será el que se localice. El algoritmo, por tanto, no está realmente diseñado para soportar elementos duplicados, aunque los tolera. Sin embargo, si se necesita una lista ordenada de elementos no duplicados, hay que usar un **TreeSet**, que se presentará más adelante en este capítulo. Éste se encarga de todos los detalles por ti automáticamente. El **TreeSet** sólo deberá ser reemplazado por un array mantenido a mano en aquellos casos en que haya cuellos de botella relacionados con el rendimiento.

Si se ha ordenado un array utilizando un **Comparador** (los arrays de tipos primitivos no permiten ordenaciones con un **Comparador**), hay que incluir el mismo **Comparador** al hacer una **binarySearch( )** (utilizando la versión sobrecargada que se proporciona de la función). Por ejemplo, el programa **OrdenarAlfabeticamente.java** puede cambiarse para que lleve a cabo una búsqueda:

```
//: c09:BuscarAlfabeticamente.java
// Buscar con un comparador.
import com.bruceeckel.util.*;
import java.util.*;

public class BuscarAlfabeticamente {
  public static void main(String[] args) {
```

```
      String[] sa = new String[30];
      Arrays2.rellenar(sa,
        new Arrays2.GeneradorStringAleatorio(5));
      ComparadorAlfabetico comp =
        new ComparadorAlfabetico ();
      Arrays.sort(sa, comp);
      int indice =
        Arrays.binarySearch(sa, sa[10], comp);
      System.out.println("Indice = " + indice);
   }
} ///:~
```

Debe pasarse el **Comparador** al **binarySearch( )** como tercer parámetro. En el ejemplo de arriba, se garantiza el éxito porque el elemento de búsqueda se ha arrancado del propio array.

Resumen de arrays

Para resumir lo visto hasta el momento, la primera y más eficiente selección a la hora de mantener un grupo de objetos debería ser un array, e incluso uno se ve obligado a seguir esta elección si lo que se desea guardar es un conjunto de datos primitivos. En el resto de este capítulo, se echará un vistazo al caso más general, en el que no se sabe en el momento de escribir el programa cuántos objetos se necesitarán o si se necesitará una forma más sofisticada de almacenamiento de los objetos. Java proporciona una biblioteca de *clases contenedoras* para solucionar este problema, en la que destacan los tipos básicos **List**, **Set** y **Map**. Utilizando estas herramientas se puede solucionar una cantidad de problemas sorprendente.

Entre sus otras características —**Set**, por ejemplo, sólo guarda un objeto de cada valor, y **Map** es un *array asociativo* que permite asociar cualquier objeto con cualquier otro— las clases contenedoras de Java redefinirán su tamaño automáticamente. Por tanto, y a diferencia de los arrays, se puede meter cualquier número de objetos y no hay que preocuparse del tamaño del contenedor cuando se está escribiendo el programa.

Introducción a los contenedores

Las clases contenedoras son una de las herramientas más potentes de cara al desarrollo puro y duro, puesto que incrementan significativamente el potencial programador. Los contenedores de Java 2 representan un rediseño[4] concienzudo de las pobres muestras de Java 1.0 y 1.1. Algunos de los rediseños han provocado mayores restricciones y precisan de un mayor cuidado. También completa la funcionalidad de la biblioteca de contenedores, proporcionando el comportamiento de las listas enlazadas y colas (con doble extremo llamadas "bicolas").

[4] Realizado Joshua Bloch en Sun.

El diseño de una biblioteca de contenedores es complicado (al igual que ocurre en la mayoría de problemas de diseño de bibliotecas). En C++, las clases contenedoras cubrían las bases con muchas clases distintas. Esto era mejor que lo que estaba disponible antes de las clases contenedoras de C++ (nada), pero no se traducía bien a Java. Por otro lado, he visto una biblioteca de contenedores consistente en una única clase, "contenedor", que actúa tanto de secuencia lineal, como de array asociativo simultáneamente. La biblioteca contenedora de Java 2 mejora el balance: se obtiene la funcionalidad completa deseada para una biblioteca de contenedores madura, y es más fácil de aprender y utilizar que las bibliotecas de clases contenedoras, y otras bibliotecas de contenedores semejantes. En ocasiones el resultado podría parecer algo extraño. A diferencia de otras decisiones hechas para las primeras bibliotecas de Java, estas extrañezas no eran accidentes, sino decisiones cuidadosamente consideradas basadas en compromisos de complejidad. Podría llevar algún tiempo adaptarse a algunos aspectos de la biblioteca, pero pienso que pronto nos haremos al uso de estas nuevas herramientas.

La biblioteca contenedora de Java 2 toma la labor de "almacenar objetos" y lo divide en dos conceptos distintos:

1. Colección **(Collection)**: grupo de elementos individuales, a los que generalmente se aplica alguna regla. Una lista **(List)** debe contener elementos en una secuencia concreta, y un conjunto **(Set)** no puede tener elementos duplicados. (Una *bolsa*, que no está implementada en la biblioteca de contenedores de Java —puesto que las **Listas** proporcionan gran parte de esta funcionalidad— no tiene estas reglas.)

2. Mapa **(Map)**: grupo de pares de objetos clave-valor. A primera vista, esto parecería una Colección **(Collection)** de pares, pero cuando se intenta implementar así, su diseño se vuelve complicado, por lo que es mejor convertirla en un concepto separado. Por otro lado, es conveniente buscar porciones de **Mapa** creando una **Colección** que la represente. Por consiguiente, un **Mapa** puede devolver un Conjunto **(Set)** de sus claves, una **Colección** de sus valores, o un **Conjunto** de sus pares. Los **Mapa**s, al igual que los arrays pueden extenderse de manera sencilla a múltiples dimensiones sin añadir nuevos conceptos: simplemente se construye un **Mapa** cuyos valores son **Mapa**s (y el valor de *esos* **Mapa**s pueden ser **Mapa**s, etc.).

Primero se echará un vistazo a las caracteríticas generales de los contenedores, después se presentarán los detalles, y finalmente se aprenderá por qué hay distintas versiones de algunos contenedores, y cómo elegir entre las mismas.

Visualizar contenedores

A diferencia de los arrays, los contenedores se visualizan elegantemente sin necesidad de ayuda. He aquí un ejemplo que muestra también los tipos básicos de contenedores:

```
//: c09:ImprimirContenedores.java
// Los contenedores se imprimen a sí mismos automáticamente.
import java.util.*;

public class ImprimirContenedores {
  static Collection rellenar(Collection c) {
```

```
      c.add("perro");
      c.add("perro");
      c.add("gato");
      return c;
   }
   static Map rellenar(Map m) {
      m.put("perro", "Bosco");
      m.put("perro", "Spot");
      m.put("gato", "Rags");
      return m;
   }
   public static void main(String[] args) {
      System.out.println(rellenar(new ArrayList()));
      System.out.println(rellenar(new HashSet()));
      System.out.println(rellenar(new HashMap()));
   }
} ///:~
```

Como se mencionó anteriormente, hay dos categorías básicas en la biblioteca de contenedores de Java. La distinción se basa en el número de elementos que se mantienen en cada posición del contenedor. La categoría **Colección** sólo mantiene un elemento en cada posición (el nombre es un poco lioso dado que a las propias bibliotecas de contenedores se les suele llamar también "colecciones"). Esta categoría incluye la **Lista**, que guarda un conjunto de elementos en una secuencia específica, y el **Conjunto**, que sólo permite la inserción de un elemento de cada tipo. La lista de Arrays (ArrayList) es un tipo de **Lista**, y el conjunto **Hash HashSet** es un tipo de **Conjunto**. Para añadir elementos a cualquier **Colección**, hay un método **add( )**.

El **Mapa** guarda pares de valores clave, de manera análoga a una mini base de datos. El programa de arriba usa una versión de **Mapa**, el **HashMap** Si se tiene un **Mapa** que asocia estados con sus capitales y se desea conocer la capital de Ohio, se mira en él —casi como si se estuviera haciendo un acceso indexado a un array. (A los **Mapa**s también se les denomina *arrays asociativos*.) Para añadir elementos a un **Mapa** hay un método **put( )** que toma una clave y un valor como argumentos. El ejemplo de arriba sólo muestra la inserción de elementos y no busca los elementos una vez añadidos éstos. Eso se mostrará más adelante.

Los métodos sobrecargados **fill( )** rellenan **Colecciones** y **Mapa**s respectivamente. Si se mira a la salida puede verse que el comportamiento impresor por defecto (proporcionado a través de los varios métodos **toString( )** de los contenedores) produce resultados bastante legibles, por lo que no es necesario un soporte adicional de impresión, como ocurre con los arrays:

```
[perro, perro, gato]
[gato, perro]
{gato=Rags, perro=Spot}
```

Una **Colección** siempre se imprime entre corchetes, separando cada elemento por comas. Un **Mapa** se imprime entre llaves, con cada clave y valor asociados mediante un signo igual (claves a la izquierda, valores a la derecha).

Se puede ver inmediatamente el comportamiento básico de cada contenedor. La **Lista** guarda los objetos exactamente tal y como se introducen, sin reordenamientos o ediciones. El **Conjunto**, sin embargo, sólo acepta uno de cada objeto y usa su propio método de ordenación interno (en general, a uno sólo le importa si algo es miembro o no del **Conjunto**, y no el orden en que aparece —para lo que se usaría una **Lista**). Y el **Mapa** sólo acepta un elemento de cada tipo, basándose en la clave, y tiene también su propia ordenación interna y no le importa el orden en que se introduzcan los elementos.

Rellenar contenedores

Aunque ya se ha resuelto el problema de impresión de los contenedores, el problema del rellenado de los mismos sufre de la misma deficiencia que **java.util.Arrays**. Exactamente igual que ocurre con **Arrays**, hay una clase denominada **Collections** que contiene métodos de utilidad **estaticos** incluyendo uno denominado **fill( )**. Este **fill( )** también se limita a duplicar una única referencia a un objeto a través de todo el contenedor, y funciona para objetos **Lista**, y no para **Conjuntos** o **Mapas**;

```
//: c09:RellenarListas.java
// El método Collections.fill().
import java.util.*;

public class RellenarListas{
  public static void main(String[] args) {
    Lista lista = new ArrayList();
    for(int i = 0; i < 10; i++)
      lista.add("");
    Collections.fill(lista, "Hola");
    System.out.println(lista);
  }
} ///:~
```

Este método es incluso menos útil de lo ya visto, debido al hecho de que sólo puede reemplazar elementos que ya se encuentran en la **Lista**, y no añadirá elementos nuevos.

Para crear ejemplos interesantes, he aquí una biblioteca complementaria **Colecciones2** (que es a su vez parte de **com.bruceeckel.util** por conveniencia) con un método **rellenar( )** (**fill( )**)que usa un generador para añadir elementos, y permite especificar el número de elementos que se desea añadir. La **interfaz Generador** definida previamente, funcionará para **Colecciones**, pero el **Mapa** requiere su propia **interfaz** generadora puesto que hay que producir un par de objetos (una clave y un valor) por cada llamada a **siguiente( )**. He aquí la clase **Par**:

```
//: com:bruceeckel:util:Par.java
package com.bruceeckel.util;
public class Par {
  public Object clave, valor;
  Par(Object k, Object v) {
    clave = k;
    valor = v;
```

```
  }
} ///:~
```

A continuación, la **interfaz** generadora que produce el **Par**:

```
//: com:bruceeckel:util:GeneradorMapa.java
package com.bruceeckel.util;
public interface GeneradorMapa {
  Par siguiente();
} ///:~
```

Con estas clases, se puede desarrollar un conjunto de utilidades para trabajar con las clases contenedoras:

```
//: com:bruceeckel:util:Colecciones2.java
// Para rellenar cualquier tipo de contenedor
// usando un objeto generador.
package com.bruceeckel.util;
import java.util.*;

public class Colecciones2 {
  // Rellenar un array usando un generador:
  public static void
  rellenar(Collection c, Generator gen, int cont) {
    for(int i = 0; i < cont; i++)
      c.add(gen.next());
  }
  public static void
  rellenar(Map m, GeneradorMapa gen, int cont) {
    for(int i = 0; i < cont; i++) {
      Par p = gen.siguiente();
      m.put(p.clave, p.valor);
    }
  }
  public static class GeneradorParStringAleatorio
  implements MapGenerator {
    private Arrays2.GeneradorParStringAleatorio gen;
    public GeneradorParStringAleatorio(int lon) {
      gen = new Arrays2.GeneradorStringAleatorio(len);
    }
    public Par siguiente() {
      return new Par(gen.next(), gen.next());
    }
  }
  // Objeto por defecto con lo que no hay que
  // crear uno propio:
```

```
public static GeneradorParStringAleatorio rsp =
  new GeneradorParStringAleatorio(10);
public static class StringPairGenerator
implements GeneradorMapa {
  private int indice = -1;
  private String[][] d;
  public GeneradorParString(String[][] datos) {
    d = datos;
  }
  public Par siguiente() {
    // Forzar que el índice sea envolvente:
    indice = (indice + 1) % d.length;
    return new Par(d[indice][0], d[indice][1]);
  }
  public GeneradorParString inicializar() {
    indice = -1;
    return this;
  }
}
// Usar un conjunto de datos predefinido:
public static GeneradorParString geografia =
  new GeneradorParString(
    CapitalesPaises.pares);
// Producir una secuencia a partir de un array 2D:
public static class GeneradorString
implements Generator {
  private String[][] d;
  private int posicion;
  private int indice = -1;
  public
  GeneradorString(String[][] datos, int pos) {
    d = datos;
    posicion = pos;
  }
  public Object siguiente() {
    // Forzar que el índice sea envolvente:
    indice = (indice + 1) % d.length;
    return d[indice][posicion];
  }
  public CcneradorString inicializar() {
    indice = -1;
    return this;
  }
}
```

```
    // Usar un conjunto de datos predefinido:
    public static GeneradorString paises =
      new GeneradorString(CapitalesPaises.pares,0);
    public static GeneradorString capitales =
      new GeneradorString(CapitalesPaises.pares,1);
} ///:~
```

Ambas versiones de **rellenar( )** toman un argumento que determina el número de *datos* a añadir al contenedor. Además, hay dos generadores para el mapa: **GeneradorParStringAleatorio**, que crea cualquier número de pares de **cadenas de caracteres** galimatías de longitud determinada por el parámetro del constructor, y **GeneradorParString**, que produce pares de **cadenas de caracteres** a partir de un array bidimensional de **String**. El **GeneradorString** también toma un array bidimensional de **cadenas de caracteres** pero genera datos simples en vez de **Pares**. Los objetos **estáticos geografía**, **países** y **capitales** proporcionan generadores preconstruidos, los últimos tres utilizando todos los países del mundo y sus capitales. Fíjese que si se intentan crear más pares de los disponibles, los generadores volverán al comienzo, y si se están introduciendo pares en **Mapa**, simplemente se ignorarían los duplicados.

He aquí el conjunto de datos predefinido, que consiste en nombres de países y sus capitales. Está escrito con fuentes de tamaño pequeño para evitar que ocupe demasiado espacio:

```
//: com:bruceeckel:util:CapitalesPaises.java
package com.bruceeckel.util;
public class CapitalesPaises {
  public static final String[][] pares = {
    // Africa
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"}, {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
    {"CHAD","N'djamena"},  {"COMOROS","Moroni"},
    {"CONGO","Brazzaville"}, {"DJIBOUTI","Dijibouti"},
    {"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"}, {"GUINEA","Conakry"},
    {"GUINEA","-"}, {"BISSAU","Bissau"},
    {"CETE D'IVOIR (IVORY COAST)","Yamoussoukro"},
    {"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
    {"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
    {"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
    {"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
    {"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
    {"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
    {"NIGER","Niamey"}, {"NIGERIA","Abuja"},
```

```
{"RWANDA","Kigali"}, {"SAO TOME E PRINCIPE","Sao Tome"},
{"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
{"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
{"SOUTH AFRICA","Pretoria/Cape Town"}, {"SUDAN","Khartoum"},
{"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
{"TOGO","Lome"}, {"TUNISIA","Tunis"},
{"UGANDA","Kampala"},
{"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)","Kinshasa"},
{"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
// Asia
{"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
{"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
{"BRUNEI","Bandar Seri Begawan"}, {"CAMBODIA","Phnom Penh"},
{"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
{"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
{"IRAN","Tehran"}, {"IRAQ","Baghdad"},
{"ISRAEL","Jerusalem"}, {"JAPAN","Tokyo"},
{"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
{"LAOS","Vientiane"}, {"LEBANON","Beirut"},
{"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
{"MONGOLIA","Ulan Bator"}, {"MYANMAR (BURMA)","Rangoon"},
{"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
{"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
{"PHILIPPINES","Manila"}, {"QATAR","Doha"},
{"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
{"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
{"SYRIA","Damascus"}, {"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
{"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
{"UNITED ARAB EMIRATES","Abu Dhabi"}, {"VIETNAM","Hanoi"},
{"YEMEN","Sana'a"},
// Australia and Oceania
{"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
{"KIRIBATI","Bairiki"},
{"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
{"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
{"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
{"PAPUA NEW GUINEA","Port Moresby"},
{"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
{"TUVALU","Fongafale"}, {"VANUATU","< Port-Vila"},
{"WESTERN SAMOA","Apia"},
// Eastern Europe and former USSR
{"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
{"BELARUS (BYELORUSSIA)","Minsk"}, {"GEORGIA","Tbilisi"},
{"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
{"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
```

```
{"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
{"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
// Europe
{"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
{"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
{"BOSNIA","-"}, {"HERZEGOVINA","Sarajevo"},
{"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
{"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
{"FINLAND","Helsinki"}, {"FRANCE","Paris"},
{"GERMANY","Berlin"}, {"GREECE","Athens"},
{"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
{"IRELAND","Dublin"}, {"ITALY","Rome"},
{"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
{"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
{"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
{"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
{"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
{"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
{"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
{"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
{"SLOVENIA","Ljujiana"}, {"SPAIN","Madrid"},
{"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
{"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
// North and Central America
{"ANTIGUA AND BARBUDA","Saint John's"}, {"BAHAMAS","Nassau"},
{"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
{"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
{"CUBA","Havana"}, {"DOMINICA","Roseau"},
{"DOMINICAN REPUBLIC","Santo Domingo"},
{"EL SALVADOR","San Salvador"}, {"GRENADA","Saint George's"},
{"GUATEMALA","Guatemala City"}, {"HAITI","Port-au-Prince"},
{"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
{"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
{"PANAMA","Panama City"}, {"ST. KITTS","-"},
{"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
{"ST. VINCENT AND THE GRENADINES","Kingstown"},
{"UNITED STATES OF AMERICA","Washington, D.C."},
// South America
{"ARGENTINA","Buenos Aires"},
{"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
{"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
{"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
```

```
      {"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
      {"PERU","Lima"}, {"SURINAME","Paramaribo"},
      {"TRINIDAD AND TOBAGO","Port of Spain"},
      {"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
   };
} ///:~
```

Esto es simplemente un array bidimensional de **cadena de caracteres**[5]. He aquí una simple prueba que utiliza los métodos **rellenar( )** y generadores:

```
//: c09:PruebaRellenar.java
import com.bruceeckel.util.*;
import java.util.*;

public class PruebaRellenar {
   static Generator sg =
      new Arrays2.GeneradorStringAleatorio(7);
   public static void main(String[] args) {
      List lista = new ArrayList();
      Colecciones2.rellenar(lista, sg, 25);
      System.out.println(lista + "\n");
      List lista2 = new ArrayLista();
      Colecciones2.rellenar(list2,
         Colecciones2.capitales, 25);
      System.out.println(lista2 + "\n");
      Set conjunto = new HashSet();
      Colecciones2.rellenar(conjunto, sg, 25);
      System.out.println(conjunto + "\n");
      Map m = new HashMap();
      Colecciones2.rellenar(m, Colecciones2.rsp, 25);
      System.out.println(m + "\n");
      Map m2 = new HashMap();
      Colecciones2.rellenar(m2,
         Colecciones2.geografia, 25);
      System.out.println(m2);
   }
} ///:~
```

Con estas herramientas se pueden probar de forma sencilla los diversos contenedores, rellenándolos con datos que interesen.

[5] Estos datos se encontraron en Internet, y después se procesaron mediante un programa en Python (véase *http://www.Python.org*).

Desventaja de los contenedores: tipo desconocido

El "inconveniente" de usar los contenedores de Java es que se pierde información de tipos cuando se introduce un objeto en un contenedor. Esto ocurre porque el programador de la clase contenedora no sabía qué tipo específico se iba a guardar en el contenedor, y construirlo de forma que sólo almacene un tipo concreto haría que éste dejase se ser una herramienta de propósito general. Así, el contenedor simplemente almacena referencias a **Object**, que es la raíz de todas las clases, y así se puede guardar cualquier tipo. (Por supuesto, esto no incluye los tipos primitivos, puesto que éstos no se heredan de nada.) Esto es una solución grandiosa excepto por:

1. Dado que se deja de lado la información de tipos al introducir un objeto en el contenedor, no hay restricción relativa al tipo de objetos que se pueden introducir en un contenedor, incluso si se desea que almacene exclusivamente, por ejemplo, gatos. Alguien podría colocar un perro sin ningún tipo de problema en ese contenedor.

2. Dado que se pierde la información de tipos, lo único que sabe el contenedor es que guarda referencias a objetos. Es necesario hacer una conversión al tipo correcto antes de usar esas referencias.

En el lado positivo, puede decirse que Java no permitirá un *uso erróneo* de los objetos que se introduzcan en un contenedor. Si se introduce un perro en un contenedor de gatos y después se intenta manipular el contenido como si de un gato se tratara, se obtendrá una excepción de tiempo de ejecución en el momento de extraer la referencia al perro del contenedor de gatos e intentar hacerle una conversión a gato.

He aquí un ejemplo utilizando el contenedor básico **ArrayList**. Los principiantes pueden pensar que **ArrayList** es "un array que se expande a sí mismo automáticamente". Usar un **ArrayList** es muy directo: se crea, se introducen objetos usando **add( )** y posteriormente se extraen con **get( )** haciendo uso del índice —exactamente igual que se haría con un array pero sin los corchetes[6].

ArrayList también tiene un método **size( )** que permite saber cuántos elementos se han añadido de forma que uno no se pasará de largo sin querer, generando una excepción.

En primer lugar, se crean las clases **Gato** y **Perro**:

```
//: c09:Gato.java
public class Gato {
  private int numGato;
  Gato(int i) { numGato = i; }
  void escribir() {
    System.out.println("Gato #" + numGato);
  }
```

[6] Éste es un punto en el que sería muy indicada la sobrecarga de operadores.

```
} ///:~


//: c09:Perro.java
public class Perro {
  private int numPerro;
  Perro(int i) { numPerro = i; }
  void escribir() {
    System.out.println("Perro #" + numPerro);
  }
} ///:~
```

Se introducen en el contenedor **Gatos** y **Perros**, y después se extraen:

```
//: c09:GatosYPerros.java
// Ejemplo simple de contenedores.
import java.util.*;

public class GatosYPerros {
  public static void main(String[] args) {
    ArrayList Gatos = new ArrayList();
    for(int i = 0; i < 7; i++)
      Gatos.add(new Gato(i));
    // Añadir perros o gatos no es ningún problema:
    Gatos.add(new Perro(7));
    for(int i = 0; i < Gatos.size(); i++)
      ((Gato)Gatos.get(i)).escribir();
    // El perro sólo se detecta en tiempo de ejecución
  }
} ///:~
```

Las clases **Gato** y **Perro** son distintas —no tienen nada en común, excepto que ambas son **Objetos**. (Si no se dice explícitamente de qué clase se está heredando, se considera que se está haciendo directamente de **Object**.) Dado que **ArrayList** guarda **Objetos**, no sólo se pueden introducir objetos **Gato** mediante el método **add( )** de **ArrayList**, sino que también es posible añadir objetos **Perro** sin que se dé ninguna queja ni en tiempo de compilación ni en tiempo de ejecución. Cuando se desea recuperar eso que se piensa que son objetos **Gato** utilizando el método **get( )** de **ArrayList**, se obtiene una referencia a un objeto que debe convertirse previamente a **Gato**. Por tanto, es necesario envolver toda la expresión con paréntesis para forzar la evaluación de la conversión antes de invocar al método **escribir( )** de **Gato**, pues si no se obtendrá un error sintáctico. Posteriormente, en tiempo de ejecución, cuando se intente convertir el objeto **Perro** en **Gato** se obtendrá una excepción.

Esto es más que sorprendente. Es algo que puede ser causa de errores muy difíciles de encontrar. Si se tiene una parte (o varias) de un programa insertando objetos en un contenedor, y se descubre mediante una expresión en un único fragmento del programa que se ha ubicado algún objeto de tipo erróneo en un contenedor, es necesario averiguar posteriormente dónde se dio la inserción errónea.

Lo positivo del asunto, es que es posible y conveniente empezar a programar con clases contenedoras estandarizadas, en vez de buscar la especificidad y complejidad del código.

En ocasiones funciona de cualquier modo

Resulta que en algunos casos parece que todo funciona correctamente sin tener que hacer una conversión al tipo original. Hay un caso bastante especial: la clase **String** tiene ayuda extra del compilador para hacer que funcione correctamente. En cualquier ocasión que el compilador espere un objeto **String** y no obtenga uno, invocará automáticamente al método **toString( )** definido en **Object** y que puede ser superpuesto por cualquier clase Java. Este método produce el objeto **String** deseado, y que es posteriormente utilizado allí donde se necesitaba un **String**.

Por tanto, todo lo que se necesita es construir objetos que superpongan el método **toString( )**, como se ve en el ejemplo siguiente:

```
//: c09:Raton.java
// Superponiendo toString().
public class Raton {
  private int numRaton;
  Raton(int i) { numRaton = i; }
  // Superponer Object.toString():
  public String toString() {
    return "Este es el Raton #" + numRaton;
  }
  public int obtenerNumero() {
    return numRaton;
  }
} ///:~
```

```
//: c09:TrabajarCualquierModo.java
// En determinados casos, las cosas simplemente
// parecen funcionar correctamente.
import java.util.*;

class TrampaRaton {
  static void capturar(Object m) {
    Raton raton = (Raton)m; // Conversión desde Object
    System.out.println("Raton: " +
      raton.obtenerNumero());
  }
}

public class TrabajarCualquierModo {
  public static void main(String[] args) {
```

```
      ArrayList ratones = new ArrayList();
      for(int i = 0; i < 3; i++)
         ratones.add(new Raton(i));
      for(int i = 0; i < ratones.size(); i++) {
         // No es necesaria conversión, se invoca automáticamente:
         // a Object.toString()
         System.out.println(
            "Raton libre: " + ratones.get(i));
         TrampaRaton.Capturar(ratones.get(i));
      }
   }
} ///:~
```

Podemos ver que en **Ratón** se ha superpuesto **toString( )**. En el segundo bucle **for** del método **main( )** se encuentra la sentencia:

```
System.out.println("Raton libre: " + ratones.get(i));
```

Después del signo "+" el compilador espera un objeto **de tipo String**. El método **get( )** produce un **Object**, por lo que para lograr la cadena de caracteres deseada, el compilador llama implícitamente a **toString( )**. Desgraciadamente, esta especie de magia sólo funciona con cadenas de caracteres: no está disponible para ningún otro tipo.

El segundo enfoque para ocultar la conversión se encuentra dentro de **TrampaRaton**. El método **capturar( )** no acepta un **Ratón** sino un **Objeto**, que después se convierte en **Ratón**. Esto es bastante presuntuoso, por supuesto, pues al aceptar un **Objeto**, se podría pasar al método cualquier cosa. Sin embargo, si la conversión es incorrecta —se pasa un tipo erróneo —se genera una excepción en tiempo de ejecución. Esto no es tan bueno como una comprobación en tiempo de compilación, pero sigue siendo robusto. Fíjese que en el uso de este método:

```
TrampaRaton.Capturar(ratones.get(i));
```

no es necesaria ninguna conversión.

Hacer un **ArrayList** consciente de los tipos

No debería abandonar aún este asunto. Una solución a toda prueba pasa por crear una nueva clase haciendo uso de **ArrayList**, que sólo acepte un determinado tipo, y produzca sólo ese determinado tipo:

```
//: c09:listaRaton.java
// Un ArrayList consciente de los tipos.
import java.util.*;

public class ListaRaton {
```

```
   private ArrayList lista = new ArrayList();
   public void aniadir(Mouse m) {
      lista.add(m);
   }
   public Raton obtener(int indice) {
      return (Raton)lista.get(indice);
   }
   public int tamanio() { return lista.size(); }
} ///:~
```

He aquí una prueba del nuevo contenedor:

```
//: c09:PruebaListaRaton.java
public class PruebaListaRaton {
   public static void main(String[] args) {
      ListaRaton mice = new ListaRaton();
      for(int i = 0; i < 3; i++)
         ratones.add(new Raton(i));
      for(int i = 0; i < ratones.size(); i++)
         TrampaRaton.capturar(ratones.get(i));
   }
} ///:~
```

Esto es similar al ejemplo anterior, excepto en que la nueva clase **ListaRaton** tiene un miembro **privado** de tipo **ArrayList**, y métodos iguales a los de **ArrayList**. Sin embargo, no acepta y produce **Objetos** genéricos, sino sólo objetos **Ratón**.

Nótese que si por el contrario se hubiera heredado **ListaRaton** de **ArrayList**, el método **aniadir(Raton)** simplemente habría sobrecargado el **add(Object)** existente, y seguiría sin haber restricción alguna en el tipo de objetos que se podrían añadir. Por consiguiente, el **ListaRaton** se convierte en un *sustituto de* **ArrayList**, que lleva a cabo algunas actividades antes de pasar la responsabilidad (véase *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*).

Dado que un objeto de tipo **ListaRaton** únicamente aceptará un **Ratón**, si se dice:

```
ratones.aniadir(new Paloma());
```

se obtendrá un mensaje de error *en tiempo de compilación*. Este enfoque, aunque es más tedioso desde el punto de vista del código, indicará inmediatamente si se está usando un tipo de manera inadecuada.

Nótese que no es necesaria ninguna conversión al usar **get( )** —siempre es un **Ratón**.

Tipos parametrizados

Este tipo de problema no está aislado —hay numerosos casos en los que es necesario crear nuevos tipos basados en otros tipos, y en los que es útil tener información de tipo específica en tiempo de compilación. Éste es el concepto de *tipo parametrizado*. En C++, esto se soporta directamente por el

lenguaje gracias a las *plantillas*. Es probable que una versión futura de Java soporte alguna variación de los tipos parametrizados; actualmente simplemente se crean clases similares a **ListaRaton**.

Iteradores

En cualquier clase contenedora, hay que tener una forma de introducir y extraer elementos. Después de todo, éste es el primer deber de un contenedor —almacenar elementos. En el **ArrayList**, **add( )** es la forma de insertar objetos, y **get( )** es *una* de las formas de extraer objetos. **ArrayList** es bastante flexible —se puede seleccionar cualquier cosa en cualquier momento, y seleccionar múltiples elementos a la vez, utilizando índices diferentes.

Si se desea empezar a pensar en un nivel superior, hay un inconveniente: hay que conocer el tipo exacto de contenedor para poder usarlo. Esto podría no parecer malo a primera vista, pero ¿qué ocurre si se empieza a usar **ArrayList**, y más adelante en el programa se descubre que debido al uso que se le está dando al contenedor sería más eficiente usar un **LinkedList** en su lugar? O suponga que se desea escribir un fragmento de código genérico independiente del tipo de contenedor con el que trabaje, ¿cómo podría hacerse de forma que éste pudiera usarse en distintos tipos de contenedores sin tener que reescribir ese código?

El concepto de *iterador* puede usarse para lograr esta abstracción. Un iterador es un objeto cuyo trabajo es moverse a lo largo de una secuencia de objetos y seleccionar cada objeto de esa secuencia sin que el programador cliente tenga que saber u ocuparse de la estructura subyacente de esa secuencia. Además, un iterador es lo que generalmente se llama un objeto "ligero": un objeto fácil de crear. Por esa razón, a menudo uno encontrará restricciones extrañas para los iteradores; por ejemplo, algunos de ellos sólo pueden moverse en una dirección.

El **Iterator** de Java es un ejemplo de un iterador con este tipo de limitaciones. No hay mucho que se pueda hacer con él salvo:

1. Pedir a un contendor que porporcione un **iterador** utilizando un método denominado **iterador ( )**. Este **Iterador** estará listo para devolver el primer elemento de la secuencia en la primera llamada a su método **next( )**.

2. Conseguir el siguiente objeto de la secuencia con **next( )**.

3. Ver si *hay* más objetos en la secuencia con **hasNext( )**.

4. Eliminar el último elemento devuelto por el iterador con **remove( )**.

Esto es todo. Es una implementación simple de un iterador, pero aún con ello es potente (y hay un **ListIterator** más sofisticado para las **Listas**). Para ver cómo funciona, podemos volver a echar un vistazo al programa **PerrosYGatos.java** visto antes en este capítulo. En la versión original, se usaba el método **get( )** para seleccionar cada elemento, pero en la versión modificada siguiente se usa un iterador:

```
//: c09:PerrosYGatos2.java
// Contenedor simple con Iterador.
import java.util.*;
```

```
public class GatosYPerros2 {
  public static void main(String[] args) {
    ArrayList gatos = new ArrayList();
    for(int i = 0; i < 7; i++)
      gatos.add(new Gato(i));
    Iterator e = gatos.iterator();
    while(e.hasNext())
      ((Gato)e.next()).escribir();
  }
} ///:~
```

Se puede ver que las últimas líneas usan ahora un **Iterador** para recorrer la secuencia, en vez de un bucle **for**. Con el **Iterador**, no hay que preocuparse por el número de elementos del contenedor. Son los métodos **hasNext( )** y **next( )** los que se encargan de esto por nosotros.

Como otro ejemplo, considérese la creación de un método de impresión de propósito general:

```
//: c09:LaberintoHamster.java
// Usando un Iterador.
import java.util.*;

class Hamster {
  private int numHamster;
  Hamster(int i) { numHamster = i; }
  public String toString() {
    return "Este el el Hamster #" + numHamster;
  }
}

class Escritura {
  static void escribirTodo(Iterador e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

public class LaberintoHamster {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 3; i++)
      v.add(new Hamster(i));
    escribir.escribirTodo(v.iterador());
  }
} ///:~
```

Echemos un vistazo a **escribirTodo( )**. Nótese que no hay información relativa al tipo de secuencia. Todo lo que se tiene es un **Iterador**, y esto es todo lo que hay que saber de la secuencia: que se puede conseguir el siguiente objeto, y que se puede saber cuándo se llega al final. Esta idea de tomar un contenedor de objetos y recorrerlo para llevar a cabo una operación sobre cada uno es potente, y se verá con detenimiento a lo largo de este libro.

El ejemplo es incluso más genérico, pues implícitamente usa el método **Object.toString( )**. El método **println( )** está sobrecargado para todos los tipos primitivos además de **Object**; en cada caso se produce automáticamente un **cadena de caracteres** llamando al método **toString( )** apropiado.

Aunque no es necesario, se puede ser más explícito usando una conversión, que tiene el efecto de llamar a **toString( )**:

```
System.out.println((String)e.next());
```

En general, sin embargo, se deseará hacer algo más que llamar a métodos de **Object**, por lo que habrá que enfrentarse de nuevo al problema de la conversión de tipos. Hay que asumir que se tiene un **Iterador** para una secuencia de un tipo particular en el que se está interesado, y que hay que convertir los objetos resultantes a ese tipo (consiguiendo una excepción en tiempo de ejecución si se hace mal).

Recursividad involuntaria

Dado que los contenedores estándar de Java son heredados (como con cualquier otra clase) de **Object**, contienen un método **toString( )**. Éste ha sido superpuesto de forma que pueden producir una representación **String** de sí mismos, incluyendo los objetos que guardan. Dentro de **ArrayList**, por ejemplo, el método **toString( )** recorre los elementos del **ArrayList** y llama a **toString( )** para cada uno de ellos. Supóngase que se desea imprimir la dirección de la clase. Parece que tiene sentido hacer simplemente referencia a **this** (son los programadores de C++, en particular, los que más tienden a esto).

```
//: c09:RecursividadInfinita.java
// Recursividad accidental.
import java.util.*;

public class RecursividadInfinita {
  public String toString() {
    return " Recursividad infinita direccion: "
      + this + "\n";
  }
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new RecursividadInfinita());
    System.out.println(v);
  }
} ///:~
```

Si simplemente se crea un objeto **RecursividadInfinita** y luego se imprime, se consigue una secuencia interminable de excepciones. Esto también es cierto si se ubican los objetos **RecursividadInfinita** en un **ArrayList** y se imprime ese **ArrayList** como se ha mostrado. Lo que está ocurriendo es una conversión de tipos automática a cadenas de caracteres. Al decir:

```
" Recursividadinfinita direccion:. " + this
```

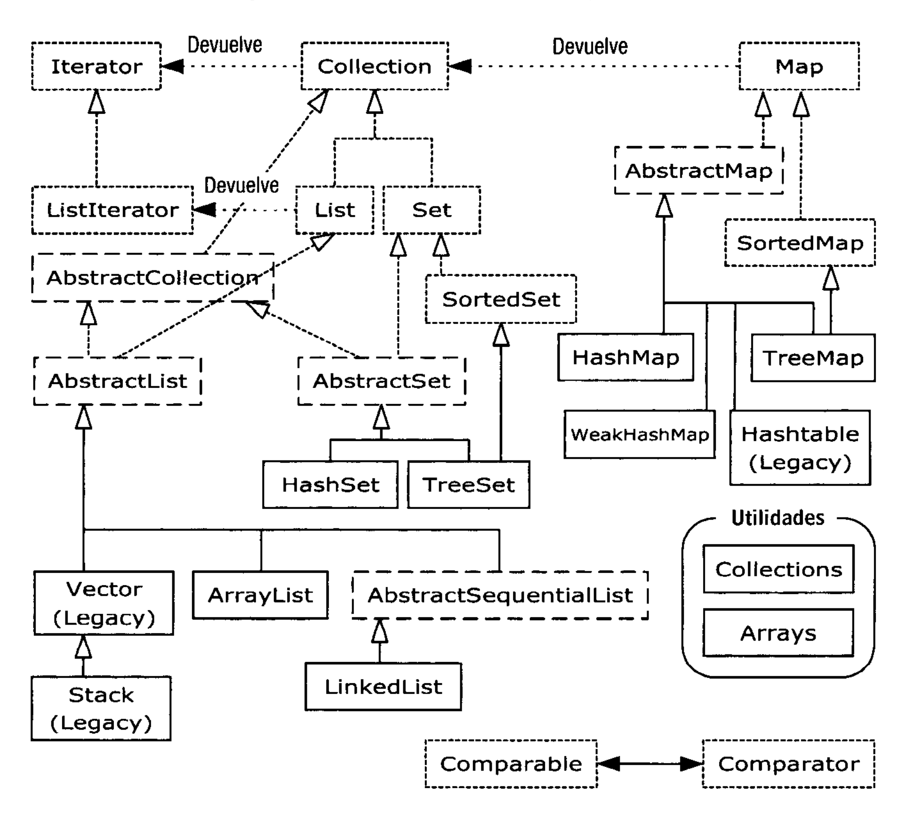
el compilador ve **cadena de carecteres** seguido de un "+" y algo que no es un **cadena de caracteres**, por lo que intenta convertir **this** a **cadena de caracteres**. Hace esta conversión llamando al método **toString( )**, lo cual produce una llamada recursiva.

Si verdaderamente se desea imprimir en este caso la dirección del objeto, la solución es llamar al método **toString( )**,de object que hace justamente eso.

Por tanto, en vez de decir **this**, se debería decir **super.toString( )**. (Esto sólo funciona si se ha heredado directamente de **Object**, o si ninguna de las clases padre ha superpuesto el método **toString( )**.)

Taxonomía de contenedores

Las **colecciones** y **mapas** pueden implementarse de distintas formas, en función de las necesidades de programación. Nos será de ayuda echar un vistazo al diagrama de los contenedores de Java 2:



Este diagrama puede ser un poco cargante al principio, pero se verá que realmente sólo hay tres componentes contenedores: **Map**, **List** y **Set**, y sólo hay dos o tres implementaciones de cada uno (habiendo una versión preferida, generalmente). Cuando vemos esto, los contenedores no son tan intimidadores.

Las cajas punteadas representan **interfaces**, las cajas a trazos representan clases **abstractas**, y las cajas continuas son clases normales (concretas). Las flechas de líneas de puntos indican que una clase particular implementa una **interfaz** (o en el caso de una clase **abstracta**, que se implementa parcialmente una **interfaz**). Las flechas continuas muestran que una clase puede producir objetos de la clase a la que apunta la flecha. Por ejemplo, cualquier **Collection** puede producir un **Iterator**, mientras que una **List**, puede producir un **ListIterator** (además de un **Iterator** normal, dado que **List** se hereda de **Collection**).

Las interfaces relacionadas con el almacenamiento de objetos son **Collection**, **List**, **Set** y **Map**. Idealmente, se escribirá la mayor parte del código para comunicarse con estas interfaces, y el único lugar en el que se especifica el tipo concreto que se está usando es en el momento de su creación. Por tanto, se puede crear un objeto **List** como éste:

```
List x = new LinkedList();
```

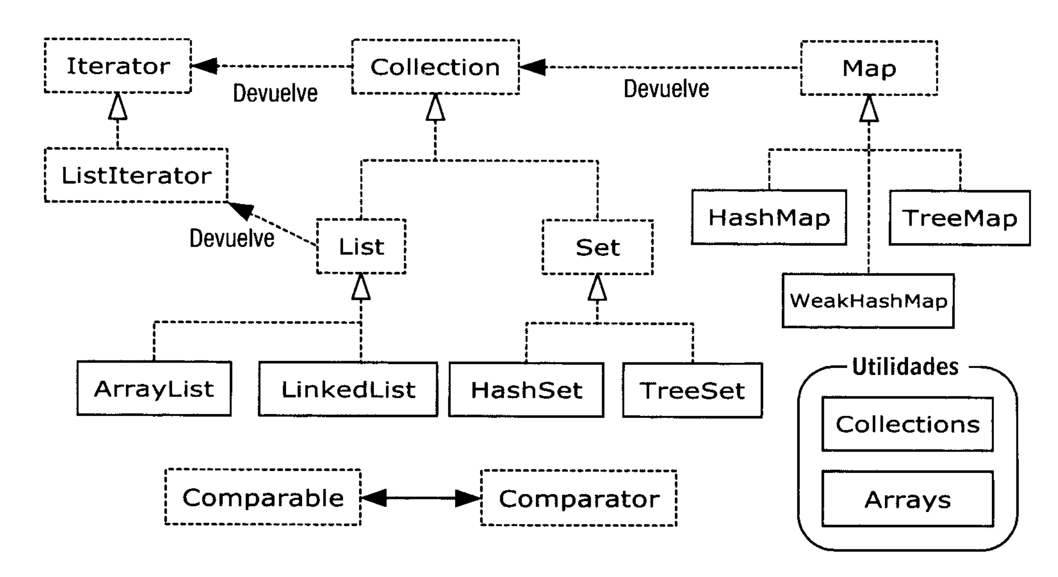
Por supuesto, se puede decidir que **x** sea una **LinkedList** (en vez de un objeto **List** genérico) y acarrear la información de tipos junto con **x**. La belleza (y la intención) de utilizar la **interfaz** es que si se desea cambiar la implementación, todo lo que hay que hacer es cambiarla en el instante de su creación, así:

```
List x = new ArrayList();
```

El resto del código puede mantenerse intacto (parte de esta generalidad puede lograrse con iteradores).

En la jerarquía de clases, se pueden ver varias clases cuyos nombres empiezan por "**Abstract**", y esto podría parecer un poco confuso al principio. Son simplemente herramientas que implementan parcialmente una interfaz particular. Si uno estuviera construyendo, por ejemplo, su propio **Set**, no empezaría con la interfaz **Set** para luego implementar todos los métodos, sino que se heredaría de **AbstractSet** haciendo el mínimo trabajo necesario para construir la nueva clase. Sin embargo, la biblioteca de contenedores contiene funcionalidad necesaria para satisfacer prácticamente todas las necesidades en todo momento. Por tanto, para nuestros propósitos, se puede ignorar cualquier clase que comience con "**Abstract**".

Por consiguiente, cuando se mire al diagrama, sólo hay que fijarse en las **interfaces** de la parte superior del diagrama y las clases concretas (las cajas de trazo continuo). Generalmente se harán objetos de clases concretas, se hará conversión hacia arriba a la **interfaz** correspondiente, y después se usará esa **interfaz** durante todo el resto del código. Además, no es necesario considerar los elementos antiguos al escribir código nuevo. Por consiguiente, el diagrama puede simplificarse enormemente a:



Ahora sólo incluye las interfaces y clases que se encontrarán normalmente, además de los elementos en los que se centrará el presente capítulo.

He aquí un ejemplo que rellena un objeto **Collection** (representado aquí con un **ArrayList**) con objetos **String**, y después imprime cada elemento del objeto **Collection**:

```
//: c09:ColeccionSencilla.java
// Un ejemplo sencillo usando las Colecciones de Java 2.
import java.util.*;

public class ColeccionSencilla {
  public static void main(String[] args) {
    // Conversión hacia arriba porque queremos
    // trabajar sólo con aspectos de Colección
    Collection c = new ArrayList();

    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator();
    while(it.hasNext())
      System.out.println(it.next());
  }
} ///:~
```

La primera línea del método **main( )** crea un objeto **ArrayList** y después hace una conversión hacia arriba a **Collection**. Dado que este ejemplo sólo usa los métodos **Collection**, funcionaría con cualquier objeto de una clase heredada de **Collection**, pero **ArrayList** es el objeto de tipo **Collection** con el que se suele trabajar.

El método **add( )**, como sugiere su nombre, pone un nuevo elemento en el objeto **Collection**. Sin embargo, la documentación establece claramente que **add( )** "asegura que este contenedor contiene el elemento especificado". Esto es para que sea compatible con el significado de **Set**, que añade el elemento sólo si no está ya ahí. Con un **ArrayList**, o cualquier tipo de **List**, **add( )** siempre significa "introducirlo", porque a las **Listas** no les importa la existencia de duplicados.

Todas las **Colecciones** pueden producir un **Iterador** mediante su método **iterator( )**. Aquí se crea un **Iterador** y luego se usa para recorrer la **Colección**, imprimiendo cada elemento.

Funcionalidad de la **Collection**

La siguiente tabla muestra todo lo que se puede hacer con una **Collection** (sin incluir los métodos que vienen automáticamente con **Object**), y por consiguiente, todo lo que se puede hacer con un **Set** o un **List**. (**List** también tiene funcionalidad adicional.) Los objetos **Map** no se heredan de **Collection**, y se tratarán de forma separada.

boolean add(Object)	Asegura que el contenedor tiene el parámetro. Devuelve falso si no añade el parámetro. (Éste es un método "opcional", descrito más adelante en este capítulo.)
boolean addAll(Collection)	Añade todos los elementos en el parámetro. Devuelve **verdadero** si se añadió alguno de los elementos. ("Opcional.")
void clear()	Elimina todos los elementos del contenedor. ("Opcional.")
boolean contains(Object)	**verdadero** si el contenedor almacena el parámetro.
boolean containsAll(Collection)	**verdadero** si el contenedor guarda todos los elementos del parámetro.
boolean isEmpty()	**verdadero** si el contenedor no tiene elementos.
Iterator iterator()	Devuelve un **Iterador** que se puede usar para recorrer los elementos del contenedor.
boolean remove(Object)	Si el parámetro está en el contenedor, se elimina una instancia de ese elemento. Devuelve **verdadero** si se produce alguna eliminación. ("Opcional.")
boolean removeAll(Collection)	Elimina todos los elementos contenidos en el parámetro. Devuelve **verdadero** si se da alguna eliminación. ("Opcional.")
boolean retainAll(Collection)	Mantiene sólo los elementos contenidos en el parámetro (una "intersección" en teoría de conjuntos). Devuelve **verdadero** si se dio algún cambio. ("Opcional.")

int size()	Devuelve el número de elementos del contenedor.
Object[] toArray()	Devuelve un array que contenga todos los elementos del contenedor.
Object[] toArray(Object[] a)	Devuelve un array que contiene todos los elementos del contenedor, cuyo tipo es el del array **a** y no un simple **Object** (hay que convertir el array al tipo correcto).

Nótese que no hay función **get( )** para selección de elementos por acceso al azar. Eso es porque **Collection** también incluye **Set**, que mantiene su propia ordenación interna (y por consiguiente convierte en carente de sentido el acceso aleatorio). Por consiguiente, si se desean examinar todos los elementos de una **Collection** hay que usar un iterador; es la única manera de recuperar las cosas.

El ejemplo siguiente demuestra todos estos métodos. De nuevo, éstos trabajan con cualquier objeto heredado de **Collection**, pero se usa un **ArrayList** como "mínimo común denominador":

```
//: c09:Coleccion1.java
// Cosas que se pueden hacer con todas las Colecciones.
import java.util.*;
import com.bruceeckel.util.*;

public class Coleccion1 {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Colecciones2.rellenar(c,
      Colecciones2.Paises, 10);
    c.add("diez");
    c.add("once");
    System.out.println(c);
    // Hacer un array a partir de Lista:
    Object[] array = c.toArray();
    // Hacer un String a partir de una Lista:
    String[] str =
      (String[])c.toArray(new String[1]);
    // Encontrar los elementos max y min; esto
    // conlleva distintas cosas en función de cómo
    // se implemente la interfaz Comparable:
    System.out.println("Collections.max(c) = " +
      Collections.max(c));
    System.out.println("Collections.min(c) = " +
      Collections.min(c));
    // Añadir una Colección a otra Colección
    Collection c2 = new ArrayList();
    Colecciones2.rellenar(c2,
```

```
      Colecciones2.Paises, 10);
    c.addAll(c2);
    System.out.println(c);
    c.remove(CapitalesPaises.pares[0][0]);
    System.out.println(c);
    c.remove(CapitalesPaises.pares[1][0]);
    System.out.println(c);
    // Quitar todos los elementos de la colección
    // pasada como parámetro:
    c.removeAll(c2);
    System.out.println(c);
    c.addAll(c2);
    System.out.println(c);
    // ¿Es un elemento de esta colección?
    String val = CapitalesPaises.pares[3][0];
    System.out.println(
      "c.contains(" + val  + ") = "
      + c.contains(val));
    // ¿Es una Colección de esta Colección?
    System.out.println(
      "c.containsAll(c2) = "+ c.containsAll(c2));
    Collection c3 = ((List)c).subList(3, 5);
    // Mantener todos los elementos que están tanto en
    // c2 como en c3 (intersección de conjuntos):
    c2.retainAll(c3);
    System.out.println(c);
    // Quitar todos los elementos
    // de c2 que también aparecen en c3:
    c2.removeAll(c3);
    System.out.println("c.isEmpty() = " +
      c.isEmpty());
    c = new ArrayList();
    Colecciones2.rellenar(c,
      Colecciones2.Paises, 10);
    System.out.println(c);
    c.clear(); // Eliminar todos los elementos
    System.out.println("despues c.clear():");
    System.out.println(c);
  }
} ///:~
```

Los objetos de tipo **ArrayList** se crean conteniendo distintos conjuntos de datos y hacen conversión hacia arriba a objetos **Collection**, por lo que está claro que no se está usando nada más que la interfaz **Collection**. El método **main( )** usa ejercicios simples para mostrar todos los métodos de **Collection**.

La sección siguiente describe las diversas implementaciones de **List**, **Set** y **Map** e indica en cada caso (con un asterisco) cuál debería ser la selección por defecto. El lector se dará cuenta de que *no* se han incluido las clases antiguas **Vector**, **Stack** y **Hashtable** porque en todos los casos son preferibles las clases Contenedoras de Java 2.

Funcionalidad de la interfaz **List**

La clase **List** básica es bastante fácil de usar, como ya se ha visto con **ArrayList**. Aunque la mayoría de veces simplemente se usará **add( )** para insertar objetos, **get( )** para sacarlos todos a la vez el **iterator( )** para lograr un **Iterador** para la secuencia, también hay un conjunto de otros métodos que podrían ser útiles.

Además, hay, de hecho, dos tipos de objetos **List**: el **ArrayList** básico que destaca entre los elementos de acceso aleatorio, y la mucho más potente **LinkedList** (que no fue diseñada para un acceso aleatorio rápido, pero que tiene un conjunto de métodos mucho más generales).

int size( )	Devuelve el número de elementos del contenedor.
List (interfaz)	El orden es la secuencia más importante de una **List**; promete mantener los elementos en una secuencia determinada. **List** añade varios métodos a **Collection** que permiten la inserción y eliminación de elementos en el medio de un objeto **List**. (Esto sólo está recomendado en el caso de **LinkedList**.) Un objeto **List** producirá un **ListIterator**, gracias al cual se puede recorrer la lista en ambas direcciones, además de insertar y eliminar elementos en medio de un objeto **List**.
ArrayList*	Un objeto **List** implementado con un array. Permite acceso aleatorio rápido a los elementos, pero es lento si se desea insertar o eliminar elementos del medio de una lista. Debería usarse **ListIterator** sólo para recorridos hacia adelante y hacia atrás de un **ArrayList**, pero no para insertar y eliminar elementos, lo que es caro si se compara con **LinkedList**.
LinkedList	Proporciona acceso secuencial óptimo, con inserciones y borrados en la parte central de la **List**. Es relativamente lenta para acceso al azar. (En este caso hay que usar **ArrayList**.) También tiene **addFirst( )**, **addLast( )**, **getFirst( )**, **getLast( )**, **removeFirst( )**, y **removeLast( )** (que no están definidos en ninguna de las interfaces o clases base) para permitir su uso como si se tratara de una pila, una cola o una bicola.

Los métodos del ejemplo siguiente cubren cada uno un grupo de actividades: las cosas que puede hacer toda lista (**basicTest( )**), recorrido con un **Iterador** (**iterMotion( )**) frente a cambiar cosas con un **Iterador** (**iterManipulation( )**), la observación de los efectos de manipulación de objetos **List** (**testVisual( )**), y operaciones disponibles únicamente para objetos **LinkedLists**.

```
//: c09:Lista1.java
// Cosas que se pueden hacer con Listas.
import java.util.*;
import com.bruceeckel.util.*;

public class Lista1 {
  public static List rellenar(List a) {
    Colecciones2.Paises.reset();
    Colecciones2.fill(a,
      Colecciones2.Paises, 10);
    return a;
  }
  static boolean b;
  static Object o;
  static int i;
  static Iterator it;
  static ListIterator lit;
  public static void PruebaBasica(List a) {
    a.add(1, "x"); // Añadir en la posición 1
    a.add("x"); // Añadir al final
    // Añadir una colección:
    a.addAll(rellenar(new ArrayList()));
    // Añadir una colección empezando en la posición 3:
    a.addAll(3, rellenar(new ArrayList()));
    b = a.contains("1"); // ¿Está ahí?
    // ¿Está la colección entera ahí?
    b = a.containsAll(rellenar(new ArrayList()));
    // Las listas permiten acceso al azar, lo que es barato
    // para ArrayList, y caro para LinkedList:
    o = a.get(1); // Recuperar el objeto de la posición 1
    i = a.indexOf("1"); // Devolver el índice de un objeto
    b = a.isEmpty(); // ¿Hay algún elemento dentro?
    it = a.iterator(); // Iterator ordinario
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Empezar en la posición 3
    i = a.lastIndexOf("1"); // Última coincidencia
    a.remove(1); // Eliminar la posición 1
    a.remove("3"); // Eliminar este objeto
    a.set(1, "y"); // Poner la posición 1 a "y"
    // Mantener todo lo que esté en los parámetros
    // (la intersección de los dos conjuntos):
    a.retainAll(rellenar(new ArrayList()));
    // Quitar todo lo que esté en los parámetros:
    a.removeAll(rellenar(new ArrayList()));
    i = a.size(); // ¿Cuán grande es?
```

```
      a.clear(); // Quitar todos los elementos
    }
    public static void movimientoIterador(List a) {
      ListIterator it = a.listIterator();
      b = it.hasNext();
      b = it.hasPrevious();
      o = it.next();
      i = it.nextIndex();
      o = it.previous();
      i = it.previousIndex();
    }
    public static void manejoIterador(List a) {
      ListIterator it = a.listIterator();
      it.add("47");
      // Moverse a un elemento después de add():
      it.next();
      // Quitar el elemento recién creado:
      it.remove();
      // Moverse a un elemento después de remove():
      it.next();
      // Cambiar el elemento recién creado:
      it.set("47");
    }
    public static void pruebaVisual(List a) {
      System.out.println(a);
      List b = new ArrayList();
      rellenar(b);
      System.out.print("b = ");
      System.out.println(b);
      a.addAll(b);
      a.addAll(rellenar(new ArrayList()));
      System.out.println(a);
      // Insertar, eliminar y reemplazar elementos
      // usando un ListIterator:
      ListIterator x = a.listIterator(a.size()/2);
      x.add("one");
      System.out.println(a);
      System.out.println(x.next());
      x.remove();
      System.out.println(x.next());
      x.set("47");
      System.out.println(a);
      // Recorrer la lista hacia atrás:
      x = a.listIterator(a.size());
      while(x.hasPrevious())
```

```
      System.out.print(x.previous() + " ");
    System.out.println();
    System.out.println("Fin de prueba Visual");
  }
  // Hay cosas que sólo pueden hacer los objetos
  // LinkedLists:
  public static void pruebaLinkedList() {
    LinkedList ll = new LinkedList();
    rellenar(ll);
    System.out.println(ll);
    // Tratarlo como una pila, apilar:
    ll.addFirst("uno");
    ll.addFirst("dos");
    System.out.println(ll);
    // Cómo "mirar a hurtadillas" a la cima de la pila:
    System.out.println(ll.getFirst());
    // Cómo desapilar de la pila:
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());
    // Tratarlo como una cola sacando elementos
    // por el final:
    System.out.println(ll.removeLast());
    // ¡Con las operaciones de arriba es una bicola!
    System.out.println(ll);
  }
  public static void main(String[] args) {
    // Hacer y rellenar una nueva lista cada vez:
    pruebaBasica(rellenar(new LinkedList()));
    pruebaBasica(rellenar(new ArrayList()));
    movimientoIterador(rellenar(new LinkedList()));
    manejoIterador(rellenar(new ArrayList()));
    manejoIterador(rellenar(new LinkedList()));
    PruebaVisual(rellenar(new ArrayList()));
    PruebaVisual(rellenar(new LinkedList()));
    pruebaLinkedList();
  }
} ///:~
```

En **pruebaBasica( )** y **movimientoIterador( )** las llamadas se hacen simplemente para mostrar la sintaxis correcta, capturando, pero no usando, el valor de retorno. En algunos casos, no se captura el valor de retorno, puesto que no suele usarse. Sería conveniente echar un vistazo a la documentación relativa a la utilización completa de cada uno de estos métodos, en la documentación *en línea* de *java.sun.com*.

Construir una pila a partir de un objeto **LinkedList**

A una pila se le suele denominar contenedor *"last-in, first-out"* o LIFO (el último en entrar es el primero en salir). Es decir, lo que se meta ("apilar") lo último en la pila, será lo primero que se puede extraer ("desapilar"). Como ocurre con el resto de contenedores de Java, lo que se mete y extrae son **Objetos**, por lo que se debe convertir lo que se extrae, a menos que se esté haciendo uso del comportamiento de **Object**.

El **LinkedList** tiene métodos que implementan directamente la funcionalidad de una pila, por lo que también se puede usar una **LinkedList** en vez de hacer una clase pila. Sin embargo, puede ser que una clase pila se comporte mejor:

```
//: c09:PilaL.java
// Haciendo una pila a partir de una LinkedList.
import java.util.*;
import com.bruceeckel.util.*;

public class PilaL {
  private LinkedList lista = new LinkedList();
  public void apilar(Object v) {
    lista.addFirst(v);
  }
  public Object cima() { return list.getFirst(); }
  public Object desapilar() {
    return lista.removeFirst();
  }
  public static void main(String[] args) {
    PilaL Pila = new PilaL();
    for(int i = 0; i < 10; i++)
      Pila.apilar(Colecciones2.Paises.next());
    System.out.println(Pila.cima());
    System.out.println(Pila.cima());
    System.out.println(Pila.desapilar());
    System.out.println(Pila.desapilar());
    System.out.println(Pila.desapilar());
  }
} ///:~
```

Si sólo se quiere el comportamiento de la pila, es inapropiado hacer uso aquí de la herencia, pues produciría una clase con todo el resto de métodos de **LinkedList** (se verá más tarde que los diseñadores de la biblioteca de Java 1.0 cometieron este error con **Stack**).

Construir una cola a partir de un objeto **LinkedList**

Una *cola* es un contenedor *"first-in, first-out"* FIFO (el primero en entrar es el primero en salir). Es decir, se introducen los elementos por un extremo y se sacan por el otro. Por tanto, los elementos se extraerán en el mismo orden en que fueron introducidos. **LinkedList** tiene métodos para soportar el comportamiento de una cola, que pueden usarse en una clase **Cola**:

```
//: c09:Cola.java
// Haciendo una cola a partir de un objeto LinkedList.
import java.util.*;

public class Cola {
  private LinkedList lista = new LinkedList();
  public void poner(Object v) { lista.addFirst(v); }
  public Object quitar() {
    return lista.removeLast();
  }
  public boolean estaVacia() {
    return lista.isEmpty();
  }
  public static void main(String[] args) {
    Cola cola = new Cola();
    for(int i = 0; i < 10; i++)
      cola.poner(Integer.toString(i));
    while(!cola.estaVacia())
      System.out.println(cola.quitar());
  }
} ///:~
```

También se puede crear fácilmente una *bicola* (cola de dos extremos) a partir de un objeto **LinkedList**. Esta es como una cola, pero se puede tanto insertar como eliminar elementos por ambos extremos de la misma.

Funcionalidad de la interfaz **Set**

Set tiene exactamente la misma interfaz que **Collection**, por lo que no hay ninguna funcionalidad como la que hay con los dos objetos **List**. Sin embargo, **Set** es exactamente igual a **Collection**; simplemente tiene un comportamiento distinto. (Éste es el uso ideal de la herencia y del polimorfismo: expresar comportamiento distinto.) Un **Set** es un objeto **Collection** que rehúsa guardar más de una instancia de cada valor de objeto (lo que constituye el "valor" del objeto es más complejo, como veremos).

Set (interfaz)	Cada elemento que se añada al objeto **Set** debe ser único; si no es así, **Set** no añade el elemento duplicado. Los **Object**s añadidos a un **Set** deben implementar **equals( )** para establecer la unicidad de los objetos. **Set** tiene exactamente el mismo interfaz que **Collection**. La interfaz **Set** no garantiza que mantenga sus elementos en ningún orden particular.
HashSet*	En los objetos **Set** en los que el tiempo de búsqueda sea importante, los **objectos** deben definir también un **HashCode( )**.
TreeSet	Un **Set** ordenado respaldado por un árbol. De esta forma, se puede extraer una secuencia ordenada de objeto **Set**.

El ejemplo siguiente *no* muestra todo lo que se puede hacer con un objeto Set, dado que la interfaz es el misma que la de **Collection**, y así se vio en el ejemplo anterior. Lo que sí hace es demostrar el comportamiento que convierte a un objeto **Set** en único:

```
//: c09:Conjunto1.java
// Cosas que se pueden hacer con conjuntos.
import java.util.*;
import com.bruceeckel.util.*;

public class Conjunto1 {
  static Colecciones2.GeneradorString gen =
    Colecciones2.Paises;
  public static void pruebaVisual(Set a) {
    Colecciones2.rellenar(a, gen.inicializar(), 10);
    Colecciones2.rellenar(a, gen.inicializar(), 10);
    Colecciones2.rellenar(a, gen.inicializar(), 10);
    System.out.println(a); // ¡Sin repeticiones!
    // Añadir otro conjunto a éste:
    a.addAll(a);
    a.add("uno");
    a.add("uno");
    a.add("uno");
    System.out.println(a);
    // Buscar algo:
    System.out.println("a.contains(\"uno\"): " +
      a.contains("uno"));
  }
  public static void main(String[] args) {
    System.out.println("HashSet");
    pruebaVisual(new HashSet());
    System.out.println("TreeSet");
    pruebaVisual(new TreeSet());
  }
```

```
} ///:~
```

Al objeto **Set** se le añaden valores duplicados, pero al imprimirlo se verá que el objeto **Set** solamente ha aceptado una instancia de cada uno de los valores.

Al ejecutar este programa veremos que el orden mantenido por el **HashSet** es distinto del de **TreeSet**, dado que cada uno tiene una manera de almacenar los elementos a fin de localizarlos después. (**TreeSet** los mantiene ordenados, mientras que **HashSet** usa una función de *conversión de clave*, diseñada específicamente para lograr búsquedas rápidas.) Cuando uno crea sus propios tipos, debe ser consciente de que un objeto **Set** necesita una manera de mantener un orden de almacenamiento, lo que significa que hay que implementar la interfaz **Comparable**, y definir el método **compareTo( )**. He aquí un ejemplo:

```
//: c09:Conjunto2.java
// Metiendo un tipo propio en un Conjunto.
import java.util.*;

class MiTipo implements Comparable {
  private int i;
  public MiTipo(int n) { i = n; }
  public boolean equals(Object o) {
    return
      (o instanceof MiTipo)
      && (i == ((MiTipo)o).i);
  }
  public int hashCode() { return i; }
  public String toString() { return i + " "; }
  public int compareTo(Object o) {
    int i2 = ((MiTipo)o).i;
    return (i2 < i ? -1 : (i2 == i ? 0 : 1));
  }
}

public class Conjunto2 {
  public static Set rellenar(Set a, int tamanio) {
    for(int i = 0; i < tamanio; i++)
      a.add(new MiTipo(i));
    return a;
  }
  public static void prueba(Set a) {
    rellenar(a, 10);
    rellenar(a, 10); // Intentar introducir duplicados
    rellenar(a, 10);
    a.addAll(rellenar(new TreeSet(), 10));
    System.out.println(a);
  }
```

```
   public static void main(String[] args) {
      prueba(new HashSet());
      prueba(new TreeSet());
   }
} ///:~
```

La forma de las definiciones de **equals( )** y **HashCode( )** se describirá más adelante en este mismo capítulo. Hay que definir un **equals( )** en ambos casos, pero el **HashCode( )** es absolutamente necesario sólo si se ubicara la clase en un **Hash.Set** (lo que es probable, pues ésta suele ser la primera opción a la hora de implementar un **Set**). Sin embargo, como estilo de programación debería superponerse **HashCode( )** siempre al superponer **equals( )**. Analizaremos el proceso detalladamente más adelante en este capítulo.

En el método **compareTo( )**, fíjese que *no* se usó la forma "simple y obvia" **return i-i2**. Aunque éste es un error de programación común, sólo funcionaría correctamente si **i** y **i2** fueran **enteros** "sin signo" (si Java *tuviera* una palabra clave "unsigned", pero no la tiene). No funciona para el **entero** con signo de Java, ya que no es lo suficientemente grande como para representar la diferencia de dos **enteros** con signo. Si **i** es un entero grande positivo y **j** es un entero grande negativo, **i-j** causará un desbordamiento y devolverá un error negativo, lo cual no funcionará.

Conjunto ordenado (SortedSet)

Si se tiene un objeto **SortedSet** (del que sólo está diponible el **TreeSet**), se garantiza que los elementos se ordenarán permitiendo proporcionar funcionalidad adicional con estos métodos de la interfaz **SortedSet**:

Comparator comparator( ): Devuelve un objeto **Comparator** utilizado para este objeto **Set**, o **null** en el caso de ordenamiento natural.

Object first( ): Devuelve el elemento menor.

Object last( ): Devuelve el elemento mayor.

SortedSet subSet(desdeElemento, hastaElemento): Proporciona una vista de este objeto **Set** desde el elemento de **desdeElemento**, incluido, hasta **hastaElemento**, excluido.

SortedSet headSet(hastaElemento): Devuelve una vista de este objeto **Set** con los elementos menores a **hastaElemento**.

SortedSet tailSet(desdeElemento): Devuelve una vista de este objeto **Set** con elementos mayores o iguales a **desdeElemento**.

Funcionalidad Map

Un objeto **ArrayList** permite hacer una selección a partir de una secuencia de objetos usando un número, por lo que en cierta forma asigna números a los objetos. Pero, ¿qué ocurre si se desea se-

leccionar un objeto de una secuencia siguiendo algún otro criterio? Una pila es un ejemplo de esto: su criterio de selección es "el último elemento insertado en la pila". Un giro contudente de esta idea de "seleccionar a partir de una secuencia" se le denomina un *mapa*, un diccionario o un *array asociativo*. Conceptualmente, parece un objeto **ArrayList**, pero en vez de buscar los objetos usando un número, se buscan utilizando ¡*otro objeto*! Éste suele ser un proceso clave en un programa.

Este concepto se presenta en Java como la interfaz **Map**. El método **put(Object clave, Object valor)** añade un valor (el elemento deseado), y le asocia una clave (el elemento con el que buscar). **get(Object clave)** devuelve el valor a partir de la clave correspondiente. También se puede probar un **Map** para ver si contiene una clave o valor con **containsKey( )** y **containsValue( )**.

La biblioteca estándar de Java tiene dos tipos diferentes de **objetos Map**: **HashMap** y **TreeMap**. Ambos tienen la misma interfaz (dado que ambos implementan **Map**), pero difieren claramente en la eficiencia. Si se observa lo que hace un **get( )**, parecerá bastante lento hacerlo buscando a través de la clave (por ejemplo) de un **ArrayList**. Es aquí donde un **HashMap** acelera considerablemente las cosas. En vez de hacer una búsqueda lenta de la clave, usa un valor especial denominado *código de tipo hash*. Ésta es una manera de tomar cierta información del objeto en cuestión y convertirlo en un **entero** "relativamente único" para ese objeto. Todos los objetos de Java pueden producir un código de tipo hash, y **HashCode( )** es un método de la clase raíz **Object**. Un **HashMap** toma un **hashCode( )** del objeto y lo utiliza para localizar rápidamente la clave. Esto redunda en una mejora dramática de rendimiento[7].

Map	Mantiene asociaciones clave-valor (pares), de forma que se puede buscar un valor usando una clave.
HashMap*	La implementación basada en una tabla de tipo hash. (Utilizar esto en vez de **Hashtable**.) Proporciona rendimiento constante en el tiempo para insertar y localizar pares. El rendimiento se puede ajustar mediante constructores que permiten especificar la *capacidad* y el *factor de carga* de la tabla de tipo hash.
TreeMap	Implementación basada en un árbol. Cuando se vean las claves de los pares, se ordenarán (en función de **Comparable** o **Comparator**, que se discutirán más tarde). La clave de un **TreeMap** es que se logran resultados en orden. **TreeMap** es el único objeto **Map** con un método **subMap( )**, que permite devolver un fragmento de árbol.

En ocasiones también es necesario conocer los detalles de cómo funciona la conversión hash, por lo que le echaremos un vistazo un poco después.

El ejemplo siguiente usa el método **Colecciones2.rellenar( )** y los conjuntos de datos de prueba definidos previamente:

---

[7] Si estas mejoras de velocidad siguen sin ajustarse a las necesidades de rendimiento pretendidas, se puede acelerar aún más la búsqueda en la tabla escribiendo un **Mapa** personalizado, y adaptándolo a tipos particulares que eviten retrasos debidos a conversiones hacia y desde **Object**. Para llegar a niveles de rendimiento aún mejores, los entusiastas de la velocidad pueden usar el *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*, de Donald Knuth, para reemplazar las listas de *cubos* de recorrido por arrays que presenten dos beneficios adicionales: pueden optimizarse para características de almacenamiento de disco y pueden ahorrar la mayoría del tiempo de creación y recolección de basura de registros individuales.

```
//: c09:Mapa1.java
// Cosas que se pueden hacer con Mapas.
import java.util.*;
import com.bruceeckel.util.*;

public class Mapa1 {
  static Colecciones2.GeneradorParString geo =
    Colecciones2.geografia;
  static Colecciones2.GeneradorParStringAleatorio
    rsp = Colecciones2.rsp;
  // Produciendo un conjunto de claves:
  public static void escribirClaves(Map m) {
    System.out.print("Tamaño = " + m.size() +", ");
    System.out.print("Claves: ");
    System.out.println(m.keySet());
  }
  // Produciendo una Colección de valores:
  public static void escribirValores(Map m) {
    System.out.print("Valores: ");
    System.out.println(m.values());
  }
  public static void prueba(Map m) {
    Colecciones2.rellenar(m, geo, 25);
    // Mapa tiene comportamiento 'de conjunto' para las claves:
    Colecciones2.rellenar(m, geo.reset(), 25);
    escribirClaves(m);
    escribirValores(m);
    System.out.println(m);
    String clave = CapitalesPaises.pares[4][0];
    String valor = CapitalesPaises.pares[4][1];
    System.out.println("m.containsKey(\"" + clave +
      "\"): " + m.containsKey(clave));
    System.out.println("m.get(\"" + clave + "\"): "
      + m.get(clave));
    System.out.println("m.containsValue(\""
      + valor + "\"): " +
      m.containsValor(valor));
    Map m2 = new TreeMap();
    Coleciones2.rellenar(m2, rsp, 25);
    m.putAll(m2);
    escribirClaves(m);
    Clave = m.keySet().iterator().next().toString();
    System.out.println("Primera clave del mapa: "+clave);
    m.remove(Clave);
    escribirClaves(m);
```

```
        m.clear();
        System.out.println("m.isEmpty(): "
          + m.isEmpty());
        Colecciones2.rellenar(m, geo.inicializar(), 25);
        // Las operaciones en el Conjunto cambian el Mapa:
        m.keySet().removeAll(m.keySet());
        System.out.println("m.isEmpty(): "
          + m.isEmpty());
    }
    public static void main(String[] args) {
        System.out.println("Prueba HashMap");
        prueba(new HashMap());
        System.out.println("Prueba TreeMap");
        prueba(new TreeMap());
    }
} ///:~
```

Los métodos **escribirClaves( )** y **escribirValores( )** no son simples utilidades, sino que también demuestran cómo producir vistas **Collection** de un **Mapa**. El método **keySet( )** devuelve un **Conjunto** respaldado por las claves de un **Mapa**. A **values( )** se le da un tratamiento similar, al producir un objeto **Colección** que contiene todos los valores del **Mapa**. (Nótese que las claves deben ser únicas, aunque los valores pueden contener duplicados.) Dado que estas **Colecciones** están respaldadas por el **Mapa**, cualquier cambio en una **Colección** se reflejará en el **Mapa** asociado.

El resto del programa proporciona ejemplos sencillos de cada operación de **Mapa**, y prueba cada tipo de **Mapa**.

Como ejemplo de uso de un objeto **HashMap**, considérese un programa que compruebe cómo funciona el método de Java **Math.random( )**. De manera ideal, produciría una distribución perfecta de números al azar, pero para probarlo es necesario generar un conjunto de números al azar y contar cuántos caen en cada subrango. Para esto es perfecto un objeto **HashMap**, puesto que asocia objetos con objetos (en este caso, el objeto valor contiene el número producido por **Math.random( )** junto con la cantidad de veces que aparece ese número):

```
//: c09:Estadisticos.java
// Demostración sencilla de HashMap.
import java.util.*;

class Contador {
    int i = 1;
    public String toString() {
        return Integer.toString(i);
    }
}

class Estadisticos {
```

```
   public static void main(String[] args) {
      HashMap hm = new HashMap();
      for(int i = 0; i < 10000; i++) {
         // Producir un número entre 0 y 20:
         Integer r =
            new Integer((int)(Math.random() * 20));
         if(hm.containsKey(r))
            ((Counter)hm.get(r)).i++;
         else
            hm.put(r, new Contador());
      }
      System.out.println(hm);
   }
} ///:~
```

En el método **main( )**, cada vez que se genera un número aleatorio, se envuelve en un objeto **Integer** de forma que pueda usarse la referencia con el objeto **HashMap**. (No se puede usar una primitiva con un contenedor, sólo una referencia a objeto.) El método **containsKey( )** comprueba si la clave ya está en el contenedor. (Es decir, ¿se ha encontrado ya el número?) En caso afirmativo, el método **get( )** produce el valor asociado para la clave, que en este caso es un objeto **Contador**. El valor **i** del contenedor se incrementa para indicar que se ha encontrado una ocurrencia más de este número al azar en particular.

Si no se ha encontrado aún la clave, el método **put( )** ubicará un nuevo par clave —valor en el objeto **HashMap**. Desde que el **Contador** inicializa automáticamente su variable **i** a uno cuando se crea, indica la primera ocurrencia de este número al azar en concreto.

Para mostrar el **HashMap**, simplemente se imprime. El método **toString( )** de **HashMap** se mueve por todos los pares clave-valor y llama a **toString( )** para cada uno. El **Integer.toString( )** está predefinido, y se puede ver el **toString( )** para **Contador**. La salida de una ejecución (a la que se han añadido saltos de línea) es:

```
{ 19=526,  18=533,  17=460,  16=513,  15=521,  14=495,
  13=512,  12=483,  11=488,  10=487,  9=514,  8=523,
  7=497,  6=487,  5=480,  4=489,  3-509,  2=503,  1= 475,
  0=505}
```

Uno se podría plantear la necesidad de la clase **Contador**, que parece que ni siquiera tuviera la funcionalidad de la clase envoltorio **Integer**. ¿Por qué no usar **int** o **Integer**? Bien, no se puede usar un tipo primitivo **entero** porque ninguno de los contenedores puede guardar nada que no sean referencias a **Objetos**. Después de ver los contenedores, puede que las clases envoltorio comiencen a tomar un mayor significado, puesto que no se puede introducir ningún tipo primitivo en los contenedores. Sin embargo, lo único que se *puede* hacer con los envoltorios de Java es inicializarlos a un valor particular y leer ese valor. Es decir, no hay forma de cambiar un valor una vez que se ha creado un objeto envoltorio. Esto hace que el envoltorio **Integer** sea totalmente inútil para solucionar nuestro problema, por lo que nos vemos forzados a crear una nueva clase para satisfacer esta necesidad.

Mapa ordenado (Sorted Map)

Si se tiene un objeto **SortedMap** (de la que **TreeMap** es lo único disponible) se garantiza que las claves se ordenarán de manera que con los siguientes métodos de la interfaz **SortedMap** se proporcione la siguiente funcionalidad:

Comparator comparator( ): Devuelve el comparador usado para este **Mapa**, o **null** para ordenación natural.

Object firstKey( ): Devuelve la clave más baja.

Object lastKey( ): Devuelve la clave más alta.

SortedMap subMap(desdeClave, hastaClave): Produce una vista de este **Mapa** con las claves que van desde **desdeClave** incluida, hasta **hastaClave** excluida.

SortedMap headMap(hastaClave): Produce una vista de este **Mapa** con las claves menores de **hastaClave**.

SortedMap tailMap(desdeClave): Produce una vista de este **Mapa** con las claves mayores o iguales que **desdeClave**.

Hashing y códigos de hash

En el ejemplo anterior, se usó una clase de biblioteca estándar (**Integer**) como clave del objeto **HashMap**. Funcionaba bien como clave, porque tenía todo lo que necesitaba. Pero se puede caer en una trampa común con objetos **HashMap** al crear clases propias para usar como claves. Por ejemplo, considérese un sistema de predicción del tiempo que une objetos **Meteorólogo** con objetos **Predicción**. Parece bastante directo—se crean dos clases, y se usa **Meteorólogo** como clave y **Predicción** como valor:

```
//: c09:DetectorPrimavera.java
// Parece creible, pero no funciona.
import java.util.*;

class Meteorologo {
  int numeroMet;
  Meteorologo(int n) { numeroMet = n; }
}

class Prediccion {
  boolean oscurecer = Math.random() > 0.5;
  public String toString() {
    if(oscurecer)
      return "¡Seis semanas mas de invierno!";
    else
```

```
        return "primavera temprana!";
    }
}

public class DetectorPrimavera {
    public static void main(String[] args) {
        HashMap hm = new HashMap();
        for(int i = 0; i < 10; i++)
            hm.put(new Meteorologo(i), new Prediccion());
        System.out.println("hm = " + hm + "\n");
        System.out.println(
            "Buscando prediccion para Meteorologo #3:");
        Meteorologo gh = new Meteorologo(3);
        if(hm.containsKey(gh))
            System.out.println((Prediccion)hm.get(gh));
        else
            System.out.println("Clave no encontrada: " + gh);
    }
} ///:~
```

A cada **Meteorólogo** se le da un número de identidad, de forma que se puede buscar una **Predicción** en el objeto **HashMap** diciendo: "Dame la **Predicción** asociada al **Meteorólogo** número 3." La clase **Predicción** contiene un **valor lógico** que se inicializa utilizando **Math.random()**, y un **toString( )** que interpreta el resultado. En el método **main( )**, se rellena un objeto **HashMap** con objetos de tipo **Meteorólogo** y sus **Predicciones** asociadas. Se imprime el **HashMap** de forma que se puede ver que ha sido rellenada. Después, se usa un **Meteorólogo** con el número de identidad 3 para buscar la predicción de **Meterólogo** número 3 (que como puede verse debe estar en el **Mapa**).

Parece lo suficientemente simple, pero no funciona. El problema es que **Meteorólogo** se hereda de la clase raíz común **Object** (que es lo que ocurre si no se especifica una clase base, pues todas las clases se heredan en última instancia de **Object**). Es el método **hashCode( )** de **Object** el que se usa para generar el código hash de cada objeto, y por defecto, usa únicamente la dirección del objeto. Por tanto, la primera instancia de **Meteorólogo(3)** *no* produce un código de hash igual al código de hash de la segunda instancia de **Meteorólogo(3)** que se intentó usar en la búsqueda.

Se podría pensar que todo lo que se necesita hacer es escribir una superposición adecuada de **hashCode( )**. Pero seguirá sin funcionar hasta que se haya hecho otra cosa: superponer el método **equals( )** que también es parte de **Object**. Este método se usa en **HashMap** al intentar determinar si una clave es igual a alguna de las claves de la tabla. De nuevo, el **Object.equals( )** por defecto simplemente compara direcciones de objetos, por lo que un **Meteorólogo(3)** no es igual a otro **Meteorólogo(3)**.

Por tanto, para utilizar nuestras clases como claves en un objeto **HashMap**, es necesario superponer tanto **hashCode( )** como **equals( )**, como se muestra en la siguiente solución al problema descrito:

```
//: c09:DetectorPrimavera2.java
// Una clase usada como clave de un HashMap
```

```
// debe superponer hashCode() y equals().
import java.util.*;

class Meteorologo2 {
  int numeroMet;
  Meteorologo2(int n) { numeroMet = n; }
  public int hashCode() { return numeroMet; }
  public boolean equals(Object o) {
    return (o instanceof Meteorologo2)
      && (numeroMet == ((Meteorologo2)o).numeroMet);
  }
}

public class DetectorPrimavera2 {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Meteorologo2(i),new Prediccion());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Buscando una prediccion para el Meteorologo #3:");
    Meteorologo2 gh = new Meteorologo2(3);
    if(hm.containsKey(gh))
      System.out.println((Prediccion)hm.get(gh));
  }
} ///:~
```

Fíjese que este ejemplo usa la clase **Predicción** del ejemplo anterior, de forma que debe compilarse primero **DetectorPrimavera.java** o se obtendrá un error de tiempo de compilación al intentar compilar **DetectorPrimavera2.java**.

Meteorologo2.hashCode( ) devuelve el número de meteorólogo como identificador. En este ejemplo, el programador es responsable de asegurar que no existirá el mismo número de ID en dos meteorólogos. No se requiere que **hashCode( )** devuelva un identificador único (algo que se entenderá mejor algo más adelante en este capítulo), pero el método **equals( )** debe ser capaz de determinar estrictamente si dos objetos son o no equivalentes.

Incluso aunque parece que el método **equals( )** sólo hace comprobaciones para ver si el parámetro es una instancia de **Meteorólogo2** (utilizando la palabra clave **instanceof**, que se explica completamente en el Capítulo 12), **instanceof** hace, de hecho, silenciosamente una segunda comprobación, para ver si el objeto es **null**, dado que **instanceof** devuelve **falso** si el parámetro de la izquierda es **null**. Asumiendo que sea del tipo correcto y no **null**, la comparación se basa en el **numeroMet** actual. Esta vez, cuando se ejecute el programa se verá que produce la salida correcta.

Al crear tu propia clase para usar en **HashSet**, hay que prestar atención a los mismos aspectos que cuando se usa como clave en un **HashMap**.

Comprendiendo **hashCode()**

El ejemplo de arriba es sólo un inicio de cara a solucionar el problema correctamente. Muestra que si no se superponen **hashCode( )** y **equals( )** para la clave, la estructura de datos de hash **(HashSet** o **HashMap**) no podrán manejar la clave adecuadamente. Sin embargo, para lograr una solución correcta al problema hay que entender lo que está ocurriendo dentro de la estructura de datos.

En primer lugar, considérese la motivación que hay tras el proceso hash se desea buscar un objeto utilizando otro objeto. Pero se puede lograr esto con un objeto **TreeSet** o un objeto **TreeMap**. También es posible implementar tu propio **Mapa**. Para hacerlo, se suministrará el método **Map.entrySet( )** para producir un conjunto de objetos **Map.Entry**. **MPar** también se definirá como el nuevo tipo de **Map.Entry**. Para poder ubicarlo en un objeto **TreeSet**, debe implementar **equals( )** y ser **Comparable**:

```
//: c09:MPar.java
// Un Map implementado con ArrayLists.
import java.util.*;

public class MPar
implements Map.Entry, Comparable {
  Object clave, valor;
  MPar(Object k, Object v) {
    clave = k;
    valor = v;
  }
  public Object obtenerClave() { return clave; }
  public Object obtenerValor() { return valor; }
  public Object ponerValor(Object v){
    Object resultado = valor;
    valor = v;
    return resultado;
  }
  public boolean equals(Object o) {
    return clave.equals(((MPar)o).clave);
  }
  public int compareTo(Object rv) {
    return ((Comparable)clave).compareTo(
      ((MPar)rv).clave);
  }
} ///:~
```

Fíjese que a las comparaciones sólo les interesan las claves, por lo que se aceptan perfectamente valores duplicados.

El ejemplo siguiente usa un **Mapa** utilizando un par de objetos **ArrayList**:

```
//: c09:MapaLento.java
// Un Mapa implementado con ArrayList.
```

```
import java.util.*;
import com.bruceeckel.util.*;

public class MapaLento extends AbstractMap {
  private ArrayList
    claves = new ArrayList(),
    valores = new ArrayList();
  public Object put(Object clave, Object valor) {
    Object resultado = get(clave);
    if(!claves.contains(clave)) {
      claves.add(clave);
      valores.add(valor);
    } else
      valores.set(claves.indexOf(clave), valor);
    return result;
  }
  public Object get(Object clave) {
    if(!claves.contains(clave))
      return null;
    return valores.get(claves.indexOf(clave));
  }
  public Set entrySet() {
    Set entradas = new HashSet();
    Iterator
      ki = claves.iterator(),
      vi = valores.iterator();
    while(ki.hasNext())
      entradas.add(new MPar(ki.next(), vi.next()));
    return entradas;
  }
  public static void main(String[] args) {
    MapaLento m = new MapaLento();
    Colecciones2.rellenar(m,
      Colecciones2.geografia, 25);
    System.out.println(m);
  }
} ///:~
```

El método **put( )** simplemente ubica las claves y valores en los objetos de tipo **ArrayList** correspondientes. En el método **main( )** se carga un **MapaLento** y después se imprime para que se vea cómo funciona.

Esto muestra que no es complicado producir un nuevo tipo de **Mapa**. Pero como sugiere el nombre, un **MapaLento** no es muy rápido, por lo que probablemente no se usará si se dispone de alguna alternativa. El problema se da en la búsqueda de la clave: no hay orden, por lo que se usa una simple búsqueda lineal, que es la forma más lenta de buscar algo.

El único motivo de utilizar hash es la velocidad: el uso de hash permite que la búsqueda se realice rápidamente. Dado que el cuello de botella se encuentra en la velocidad de búsqueda de la clave, una de las soluciones al problema podría ser mantener las claves ordenadas y después realizar la búsqueda usando **Collections.binarySearch( )** (un ejercicio planteado al final de este capítulo consiste en esto).

El uso de hash va aún más lejos diciendo que todo lo que se desea hacer es almacenar la clave *en algún sitio,* de forma que pueda encontrarse rápidamente. Como se ha visto en este capítulo, la estructura más rápida en la que almacenar un grupo de elementos es un array, por lo que se usará éste para representar la información de claves (fíjese en el detalle de que dijimos "información de claves" y no las propias claves). También se ha visto en este capítulo que un array, una vez asignado, no se puede redimensionar, por lo que se tiene un problema: se desea poder almacenar cualquier número de valores en el **Mapa**, pero si el número de claves viene fijado por el tamaño del array ¿qué se puede hacer?

La respuesta es que el array no almacenará las claves. Desde la clave del objeto, se derivará un número que se indexará al array. Este número es el código de *hash*, producido por el método **HashCode( )** (en informática, a esta función se le denominaría *función de hash*) definido en **Object**, y presumiblemente superpuesto en la propia clase. Para solucionar el problema del array de tamaño fijo, se permite que más de una clave produzca el mismo índice. Es decir, puede haber *colisiones*. Debido a esto, no importa lo grande que sea el array, puesto que cada objeto caerá en algún lugar del mismo.

Por tanto el proceso de buscar un valor comienza computando el código de hash y usándolo como índice del array. Si se pudiera garantizar que no se dieran colisiones (lo que podría ser posible si se tuviera un número fijo de valores) entonces se tendría una *función de hash perfecta*, pero éste es un caso especial. En el resto de ocasiones, las colisiones se manejan mediante *encadenado externo*: el array no apunta directamente a un valor, sino a una lista de valores. Estos valores se buscan de manera lineal utilizando el método **equals( )**. Por supuesto, este aspecto de la búsqueda es mucho más lento, pero si la función de hash fuera buena sólo habría unos pocos valores (como máximo) en cada posición. Por tanto, en vez de buscar por toda la lista, se salta directamente a una *ranura* en la que simplemente hay que comprobar unas pocas entradas para encontrar el valor. Esto es mucho más rápido, siendo la razón la rapidez de **HashMap**.

Conociendo lo básico del uso de claves hash, es posible implementar un **Mapa** simple con técnicas de hash:

```
//: c09:HashMapSencillo.java
// Una demostración del Mapa que utiliza hash.
import java.util.*;
import com.bruceeckel.util.*;

public class HashMapSencillo extends AbstractMap {
  // Elegir un número primo para el tamaño de la tabla de
  // hash, para lograr una distribución uniforme:
  private final static int SZ = 997;
  private LinkedList[] cubo= new LinkedList[SZ];
```

```
public Object put(Object clave, Object valor) {
   Object resultado = null;
   int indice = clave.hashCode() % SZ;
   if(indice < 0) indice = -indice;
   if(cubo[indice] == null)
     cubo[indice] = new LinkedList();
   LinkedList pares = cubo[index];
   MPar par = new MPar(clave, valor);
   ListIterator it = pares.listIterator();
   boolean encontrado = false;
   while(it.hasNext()) {
     Object iPar = it.next();
     if(iPar.equals(par)) {
       resultado = ((MPar)iPar).obtenerValor();
       it.set(par); // Reemplazar viejo con nuevo
       encontrado = true;
       break;
     }
   }
   if(!encontrado)
     cubo[indice].add(par);
   return result;
}
public Object get(Object clave) {
   int indice = clave.hashCode() % SZ;
   if(indice < 0) indice = -indice;
   if(cubo[indice] == null) return null;
   LinkedList pares = cubo[indice];
   MPar parear = new MPar(clave, null);
   ListIterator it = pares.listIterator();
   while(it.hasNext()) {
     Object iPar = it.next();
     if(iPar.equals(parear))
       return ((MPar)iPar).obtenerValor();
   }
   return null;
}
public Set entrySet() {
   Set entradas = new HashSet();
   for(int i = 0; i < cubo.length; i++) {
     if(cubo[i] == null) continue;
     Iterator it = cubo[i].iterator();
     while(it.hasNext())
       entradas.add(it.next());
   }
```

```
      return entradas;
   }
   public static void main(String[] args) {
      SimpleHashMap m = new SimpleHashMap();
      Colecciones2.rellenar(m,
         Colecciones2.geografia, 25);
      System.out.println(m);
   }
} ///:~
```

Dado que a las "posiciones" de una tabla de hash se les suele llamar cubos *(buckets)*, se llama **cubo** al array que representa esta tabla. Para promocionar la distribución uniforme, el número de cubos suele ser un número primo. Fíjese que es un array de **LinkedList**, que automáticamente soporta colisiones —cada nuevo *elemento* simplemente se añade al final de la lista.

El valor de retorno de **put( )** es **null** o, si la clave ya estaba en la lista, el valor viejo asociado a esa clave. El valor de retorno es **resultado**, que se inicializa a **null**, pero si se descubre una clave en la lista, se asigna **resultado** a esa clave.

Tanto para **put( )** como **get( )**, lo primero que ocurre es que se invoca a **hashCode( )** para lograr la clave, y se fuerza a que el resultado sea positivo. Después, se hace que encaje en el array **cubo** utilizando el operador módulo y el tamaño del array. Si esa posición vale **null**, quiere decir que no hay elementos a los que el hash condujo a esa posición, por lo que se crea una nueva **LinkedList** para guardar los objetos. Sin embargo, el proceso normal es mirar en la lista para ver si hay duplicados, y si los hay, se pone el valor viejo en **resultado** y el nuevo valor reemplaza al viejo. El indicador **encontrado** mantiene un seguimiento de si se ha encontrado un par clave-valor antiguo, y si no, se añade el nuevo par al final de la lista.

En **get( )**, se verá código muy similar al contenido en **put( )**, pero más simple. Se calcula el índice en el array **cubo**, y si existe una **LinkedList** en ella, se busca la coincidencia en la misma.

El método **entrySet( )** debe encontrar y recorrer todas las listas, añadiéndolas al **Set**. Una vez que se ha creado este método, se prueba el **Map** rellenándolo de valores para después imprimirlos.

Factores de rendimiento de **HashMap**

Para entender los aspectos es necesaria cierta terminología:

> ***Capacidad***: El número de posiciones de la tabla.
>
> ***Capacidad inicial***: Número de posiciones en la creación de la tabla.
>
> **HashMap** y **HashSet**: Tienen constructores que permiten especificar su capacidad inicial.
>
> ***Tamaño***: Número de elementos actualmente en la tabla.
>
> ***Factor de carga***: tamaño/capacidad. Un factor de carga 0 es una tabla vacía; 0,5 es una tabla medio llena, etc. Una tabla con una densidad de carga alta tendrá pocas colisiones, por lo que es óptima para búsquedas e inserciones (pero ralentizará el proceso de recorrido con

un iterador). **HashMap** y **HashSet** tienen constructores que permiten especificar el factor de carga, lo que significa que cuando se llega a este factor de carga el contenedor incrementará automáticamente la capacidad (el número de posiciones) doblándola, y redistribuirá los objetos existentes en el nuevo conjunto de posiciones (a esta operación se la llama *redistribución de las claves hash*).

El factor de carga por defecto utilizado por **HashMap** es 0,75 (no hace una redistribución de claves hash hasta que 3/4 de la tabla están llenos). Esto parece ofrecer un buen equilibrio entre costo en espacio y costo en tiempo. Un factor de carga mayor disminuye el espacio que la tabla necesita, pero incrementa el coste de búsqueda, que es importante pues lo que más se hará son búsquedas (incluyendo **get( )** y **put( )**.

Si se sabe que se almacenarán muchas entradas en un **HashMap**, crearlo con una capacidad inicial apropiadamente grande, evitará la sobrecarga debida a la redistribución automática.

Superponer el método **hashCode( )**

Ahora que se entiende lo que está involucrado en la función del objeto **HashMap**, los aspectos involucrados en la escritura de método **hashCode( )** tendrán más sentido.

En primer lugar, no se tiene control de la creación del valor actual utilizado para indexar el array de elementos. Este valor depende de la capacidad del objeto **HashMap** particular, y esa capacidad cambia dependiendo de lo lleno que esté el contenedor y de cuál sea el factor de carga. El valor producido por el método **hashCode( )** se procesará después para crear el índice de los elementos (en **SimpleHashMap** el cálculo es un módulo por el tamaño del array de elementos).

El factor más importante de cara a la creación de un método **hashCode( )** es que, independientemente de cuándo se llame a **hashCode( )**, produzca el mismo valor para cada objeto cada vez que se le llame. Si se tuviera un objeto que produce un valor **hashCode( )** cuando se invoca a **put( )** para introducirlo en un **HashMap**, y otro durante un **get( )**, no se podrían retirar los objetos. Por tanto, si el **hashCode( )** depende de datos mutables del objeto, el usuario debe ser consciente de que cambiar los datos producirá una clave diferente generando un **hashCode( )** distinto.

Además, probablemente *no* se desee generar un **hashCode( )** que se base en una única información de un objeto —en particular, el valor de **this** genera un **hashCode( )** malo, pues no se puede generar una nueva clave idéntica a la usada al hacer **put( )** con el par clave-valor original. Éste era el problema de **DetectorPrimavera.java**, dado que la implementación por defecto de **hashCode( )** *sí* que utiliza la dirección del objeto. Por tanto, se deseará utilizar información del objeto que lo identifique de manera significativa.

En la clase **String**, puede verse un ejemplo de esto. Los objetos **String** tienen la característica especial de que si un programa tiene varios objetos **String** con secuencias de caracteres idénticas, todos esos objetos **String** hacen referencia a la misma memoria (el mecanismo de esto se describe en el Apéndice A). Por tanto, tiene sentido que el método **hashCode( )** producido por dos instancias distintas de **new String("hola")** debería ser idéntico. Esto se puede comprobar ejecutando el siguiente programa:

```
//: c09:StringHashCode.java
```

```
public class StringHashCode {
  public static void main(String[] args) {
    System.out.println("Hola".hashCode());
    System.out.println("Hola".hashCode());
  }
} ///:~
```

Para que esto funcione, el método **hashCode( )** de **String** debe basarse en los contenidos de **String**.

Por tanto para que un método **hashCode( )** sea efectivo, debe ser rápido y significativo: es decir, debe generar un valor basado en los contenidos del objeto. Recuérdese que este valor no tiene por qué ser único —se debería preferir la velocidad más que la unicidad —pero entre **hashCode( )** y **equals( )** debería resolverse completamente la identidad del objeto.

Dado que **hashCode( )** se procesa antes de producir el índice de elementos, el rango de valores no es importante; simplemente es necesario generar un valor **entero**.

Hay aún otro factor: un buen método **hashCode( )** debería producir una distribución uniforme de los valores. Si los valores tienden a concentrarse en una zona, el objeto **HashMap** o el objeto **HashSet** estarán más cargados en algunas áreas y no serían tan rápidos como podrían ser con una función de hashing que produzca distribuciones uniformes.

He aquí un ejemplo que sigue estas líneas:

```
//: c09:CuentaString.java
// Creando un buen hashCode().
import java.util.*;

public class CuentaString {
  private String s;
  private int id = 0;
  private static ArrayList creado =
    new ArrayList();
  public CuentaString(String str) {
    s = str;
    creado.add(s);
    Iterator it = creado.iterator();
    // Id es el número total de instancias
    // de este string en uso por CuentaString:
    while(it.hasNext())
      if(it.next().equals(s))
        id++;
  }
  public String toString() {
    return "String: " + s + " id: " + id +
      " hashCode(): " + hashCode() + "\n";
  }
```

```
  public int hashCode() {
    return s.hashCode() * id;
  }
  public boolean equals(Object o) {
    return (o instanceof CuentaString)
      && s.equals(((CuentaString)o).s)
      && id == ((CuentaString)o).id;
  }
  public static void main(String[] args) {
    HashMap m = new HashMap();
    CuentaString[] cs = new CuentaString[10];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CuentaString("hi");
      m.put(cs[i], new Integer(i));
    }
    System.out.println(m);
    for(int i = 0; i < cs.length; i++) {
      System.out.print("Buscando " + cs[i]);
      System.out.println(m.get(cs[i]));
    }
  }
} ///:~
```

CuentaString incluye un **String** y un **id** que representa el número de objetos **CuentaString** que contienen un **String** idéntico. El conteo lo lleva a cabo el constructor iterando a través del método estatico **ArrayList** donde se almacenan todos los **Strings**.

Tanto **hashCode( )** como **equals( )** producen resultados basados en ambos campos; si se basara simplemente en un **String** o en el **id** habría coincidencias duplicadas para valores distintos.

Fíjese lo simple que es **hashCode( )**: el **hashCode( )** del **String** se multiplica por el **id**. Cuanto más pequeño sea **hashCode( )**, mejor (y más rápido).

En el método **main( )** se crea un conjunto de objetos **CuentaString**, utilizando el mismo **String** para mostrar que los duplicados crean valores únicos debido a **id**. Se muestra el **HashMap,** de forma que se puede ver cómo se almacena internamente (en un orden imperceptible) y se busca cada clave individualmente para demostrar que el mecanismo de búsqueda funciona correctamente.

Guardar referencias

La biblioteca **java.lang.ref** contiene un conjunto de clases que permiten mayor flexibilidad en la recolección de basura, siendo especialmente útiles cuando se tiene objetos grandes que pueden agotar la memoria. Hay tres clases heredadas de la clase abstracta **Reference**: **SoftReference**, **WeakReference**, y **PhantomReference**. Cada una proporciona un nivel de direccionamiento diferente por parte del recolector de basura, si el objeto en cuestión *sólo* es alcanzable a través de uno de estos objetos **Reference**.

El que un objeto sea *accesible* significa que en algún sitio del programa se puede encontrar el objeto. Esto podría significar que se tiene una referencia en la pila que va directa al objeto, pero también se podría tener una referencia a un objeto que tiene una referencia al objeto en cuestión; podría haber muchos enlaces intermedios. Si un objeto es accesible, el recolector de basura no puede liberar el espacio que usa porque sigue en uso por parte del programa. Si no se puede acceder a un objeto, no hay forma de que el programa lo use, por lo que es seguro que el recolector lo eliminará.

Se usan objetos **Reference** cuando se desea seguir guardando una referencia a ese objeto —se desea ser capaz de acceder a ese objeto— pero también se quiere permitir al recolector de basura liberar ese objeto. Por consiguiente, se tiene alguna forma de usar el objeto, pero si se está cerca de una saturación de la memoria, se permite la recolección del objeto.

Esto se logra usando un objeto **Reference** que es un intermediario entre el programador y la referencia ordinaria, *y* no debe haber referencias ordinarias al objeto (las no envueltas dentro de objetos **Reference**). Si el recolector de basura descubre que se puede acceder a un objeto mediante una referencia ordinaria, no liberará ese objeto.

Si se ordenan **SoftReference**, **WeakReference**, y **PhantomReference**, cada uno es más "débil" que el anterior en lo que a nivel de "accesibilidad" se refiere. Las referencias blandas son para implementar cachés sensibles. Las referencias débiles son para implementar "correspondencias canónicas" —en las que las instancias de los objetos pueden usarse simultáneamente en múltiples lugares del programa, para ahorrar espacio de almacenamiento— que no evitan que sus claves (o valores) puedan ser reclamadas. Las referencias fantasma (*phantom*) permiten organizar acciones de limpieza antes de su muerte de forma más flexible que lo que permite el mecanismo de finalización de Java.

Con **SoftReference**s y **WeakReference**s se puede elegir si ubicarlas en una **ReferenceQueue** (el dispositivo usado para acciones de limpieza antes de su muerte), pero sólo se puede construir una **PhantomReference** en una **ReferenceQueue**. He aquí una demostración:

```
//: c09:Referencias.java
// Demuestra los objetos Referencia
import java.lang.ref.*;

class MuyGrande {
  static final int SZ = 10000;
  double[] d = new double[SZ];
  String ident;
  public MuyGrande(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizando " + ident);
  }
}

public class Referencias {
  static ReferenceQueue rq= new ReferenceQueue();
```

```
  public static void comprobarCola() {
    Object inq = rq.poll();
    if(inq != null)
      System.out.println("In cola: " +
        (MuyGrande)((Reference)inq).get());
  }
  public static void main(String[] args) {
    int tamanio = 10;
    // O, elegir el tamaño a través de la línea de comandos:
    if(args.length > 0)
      tamanio = Integer.parseInt(args[0]);
    SoftReference[] sa =
      new SoftReference[tamanio];
    for(int i = 0; i < sa.length; i++) {
      sa[i] = new SoftReference(
        new MuyGrande("Blando " + i), rq);
      System.out.println("Recien creado: " +
        (MuyGrande)sa[i].get());
      comprobarCola();
    }
    WeakReference[] wa =
      new WeakReference[tamanio];
    for(int i = 0; i < wa.length; i++) {
      wa[i] = new WeakReference(
        new MuyGrande("debil " + i), rq);
      System.out.println("Recien creado: " +
        (MuyGrande)wa[i].get());
      comprobarCola();
    }
    SoftReference s = new SoftReference(
      new MuyGrande("Blando"));
    WeakReference w = new WeakReference(
      new MuyGrande("Debil"));
    System.gc();
    PhantomReference[] pa =
      new PhantomReference[tamanio];
    for(int i = 0; i < pa.length; i++) {
      pa[i] = new PhantomReference(
        new MuyGrande("Fantasma " + i), rq);
      System.out.println("Recien creado: " +
        (MuyGrande)pa[i].get());
      comprobarCola();
    }
  }
} ///:~
```

Cuando se ejecute este programa (se querrá encauzar la salida a través de una utilidad "más" de forma que se pueda ver la salida en varias páginas), se verá que se recolectan todos los objetos, incluso aunque se siga teniendo acceso a los mismos a través del objeto **Reference** (para conseguir la referencia al objeto actual, se usa **get( )**). También se verá que **ReferenceQueue** siempre produce un objeto **Reference** que contiene un objeto **null**. Para hacer uso de esto, se puede heredar de la clase **Reference** concreta en la que se esté interesado y añadir más métodos útiles al nuevo tipo **Reference**.

El objeto HasMap débil **(WeakHashMap)**

La biblioteca de contenedores tiene un **Mapa** especial para guardar referencias débiles: el **WeakHashMap**. Esta clase se diseña para facilitar la creación de correspondencias canónicas. En este tipo de correspondencias, se ahorra espacio de almacenamiento haciendo sólo una instancia de un valor particular. Cuando el programa necesita ese valor, busca el objeto existente en el mapa y lo usa (en vez de crearlo de la nada). La correspondencia puede hacer los valores como parte de esta inicialización, pero es más probable que los valores se hagan bajo demanda.

Dado que esta técnica permite ahorrar espacio de almacenamiento, es muy conveniente que el **WeakHashMap** permita al recolector de basura limpiar automáticamente las claves y valores. No se tiene que hacer nada especial a las claves y valores a ubicar en el **WeakHashMap**; éstos se envuelven automáticamente en **WeakReferences** por parte del mapa. El disparador para permitir la limpieza es que la clave deje de estar en uso, como aquí se demuestra:

```
//: c09:MapeoCanonico.java
// Demuestra WeakHashMap.
import java.util.*;
import java.lang.ref.*;

class Clave {
  String ident;
  public Clave(String id) { ident = id; }
  public String toString() { return ident; }
  public int hashCode() {
    return ident.hashCode();
  }
  public boolean equals(Object r) {
    return (r instanceof Clave)
      && ident.equals(((Clave)r).ident);
  }
  public void finalize() {
    System.out.println("Finalizando la Clave "+ ident);
  }
}

class Valor {
  String ident;
```

```
  public Valor(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizando el Valor "+ident);
  }
}

public class MapeoCanonico {
  public static void main(String[] args) {
    int tamanio = 1000;
    // O, elegir el tamaño mediante la línea de comandos:
    if(args.length > 0)
      tamanio = Integer.parseInt(args[0]);
    Clave[] claves = new Clave[tamanio];
    WeakHashMap whm = new WeakHashMap();
    for(int i = 0; i < size; i++) {
      Clave k = new Clave(Integer.toString(i));
      Valor v = new Valor(Integer.toString(i));
      if(i % 3 == 0)
        claves[i] = k; // Salvar como referencias "reales"
      whm.put(k, v);
    }
    System.gc();
  }
} ///:~
```

La clase **Clave** debe tener un método **hashCode( )** y un método **equals( )** dado que se está usando como clave en una estructura de datos con tratamiento hash, como se describió previamente en este capítulo.

Cuando se ejecute el programa se verá que el recolector de basura se saltará la tercera clave, pues también se ha ubicado esa clave en el array **claves** y, por consiguiente, estos objetos no podrán ser recolectados.

Revisitando los iteradores

Ahora se puede demostrar la verdadera potencia del **Iterador**: la habilidad de separar la operación de recorrer una secuencia de la estructura subyacente de esa secuencia. En el ejemplo siguiente, la clase **EscribirDatos** usa un **Iterador** para recorrer una secuencia y llama al método **toString( )** para cada objeto. Se crean dos tipos de contenedores distintos —un **ArrayList** y un **HashMap—** y se rellenan respectivamente con objetos **Ratón** y **Hamster**. (Estas clases se definen anteriormente en este capítulo.) Dado que un **Iterador** esconde la estructura del contenedor subyacente, **EscribirDatos** no sabe o no le importa de qué tipo de contenedor proviene el **Iterador**:

```
//: c09:Iteradores2.java
// Revisitando los Iteradores.
```

```
import java.util.*;

class EscribirDatos {
   static void escribir(Iterator e) {
      while(e.hasNext())
         System.out.println(e.next());
   }
}

class Iteradores2 {
   public static void main(String[] args) {
      ArrayList v = new ArrayList();
      for(int i = 0; i < 5; i++)
         v.add(new Raton(i));
      HashMap m = new HashMap();
      for(int i = 0; i < 5; i++)
         m.put(new Integer(i), new Hamster(i));
      System.out.println("ArrayList");
      EscribirDatos.print(v.iterator());
      System.out.println("HashMap");
      EscribirDatos.print(m.entrySet().iterator());
   }
} ///:~
```

Para el objeto **HashMap como**, el método **entrySet( )** produce un **conjunto** de objetos **Map.entry**, que contiene tanto la clave como el valor de cada entrada, por lo que se visualizan los dos.

Fíjese que **EscribirDatos.escribir( )** saca ventaja del hecho de que estos contenedores son de clase **Object**, por lo que la llamada a **toString( )** por parte de **System.out.println( )** es automática. Es más probable que en un problema haya que asumir que un **Iterador** esté recorriendo un contenedor de algún tipo específico. Por ejemplo, se podría imaginar que lo que contiene el contenedor es un **Polígono** con un método **dibujar( )**. Entonces habría que hacer una conversión hacia abajo del **Object** devuelto por **Iterator.next( )** para producir un **Polígono**.

Elegir una implementación

Hasta ahora, debería haberse entendido que sólo hay tres componentes contenedores: **Map**, **List** y **Set**, y sólo dos o tres implementaciones de cada interfaz. Si se necesita la funcionalidad ofrecida por una **interfaz** particular ¿cómo se decide qué implementación particular usar?

Para entender la respuesta, hay que ser consciente de que cada implementación diferente tiene sus propias caracteríticas, ventajas y desventajas. Por ejemplo, se puede ver en el diagrama que la "característica" de **Hashtable**, **Vector**, y **Stack** es que son clases antiguas, de forma que el código antiguo sigue funcionando. Por otro lado, es mejor si no se utilizan para código nuevo (Java 2).

La diferencia entre los otros contenedores suele centrarse en aquello por lo que están "respaldados"; es decir, las estructuras de datos que implementan físicamente la **interfaz** deseada. Esto significa que, por ejemplo, **ArrayList** y **LinkedList** implementan la interfaz **List**, de forma que el programa producirá los mismos resultados independientemente del que se use.

Sin embargo, un **ArrayList** está respaldado por un array, mientras que el **LinkedList** está implementado de la forma habitual de una lista doblemente enlazada, como objetos individuales cada uno de los cuales contiene datos junto con referencias a los elementos previo y siguiente de la lista. Debido a esto, si se desean hacer muchas inserciones y retiradas en la parte central de la lista, la selección apropiada es **LinkedList**. (**LinkedList** también tiene funcionalidad adicional establecida en **AbstractSequentialList**). Si no, suele ser más rápido un **ArrayList**.

Como otro ejemplo, se puede implementar un objeto **Set,** bien como un **TreeSet** o bien como **HashSet**. Un **TreeSet** está respaldado por un **TreeMap** y está diseñado para producir un conjunto ordenado. Sin embargo, si se va a tener cantidades mayores de datos en el **Set**, el rendimiento de las inserciones de **TreeSet** decaerá. Al escribir un programa que necesite un **Set**, debería elegirse **HashSet** por defecto, y cambiar a **TreeSet** cuando es más importante tener un conjunto constantemente ordenado.

Elegir entre **Listas**

La manera más convincente de ver las diferencias entre las implementaciones de **List** es con una prueba de rendimiento. El código siguiente establece una clase base interna que se usa como sistema de prueba, después crea un array de clases internas anónimas, una para cada prueba. El método **probar( )** llama a cada una de estas clases internas. Este enfoque te permite insertar y eliminar sencillamente nuevos tipos de pruebas.

```
//: c09:RendimientoListas.java
// Demuestra diferencias de rendimiento en Listas.
import java.util.*;
import com.bruceeckel.util.*;

public class RendimientoListas {
  private abstract static class Realizar Prueba {
    String nombre;
    int tamanio; // Cantidad de pruebas
    RealizarPrueba(String nombre, int tamanio) {
      this.nombre = nombre;
      this.tamanio = tamanio;
    }
    abstract void probar(List a, int reps);
  }
  private static RealizarPruebas[] Pruebas = {
    new RealizarPruebas("get", 300) {
      void probar(List a, int reps) {
```

```
        for(int i = 0; i < reps; i++) {
          for(int j = 0; j < a.tamanio(); j++)
            a.get(j);
        }
      }
    },
    new RealizarPrueba("iteracion", 300) {
      void probar(List a, int reps) {
        for(int i = 0; i < reps; i++) {
          Iterator it = a.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
    new RealizarPrueba("insercion", 5000) {
      void probar(List a, int reps) {
        int mitad = a.size()/2;
        String s = "test";
        ListIterator it = a.listIterator(mitad);
        for(int i = 0; i < tamanio * 10; i++)
          it.add(s);
      }
    },
    new RealizarPrueba("eliminacion", 5000) {
      void probar(List a, int reps) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext()) {
          it.next();
          it.remove();
        }
      }
    },
  };
  public static void probar(List a, int reps) {
    // Un truco para imprimir el nombre de la clase:
    System.out.println("Probando " +
      a.getClass().getName());
    for(int i = 0; i < pruebas.length; i++) {
      Colecciones2.rellenar(a,
        Colecciones2.paises.inicializar(),
        pruebas[i].tamanio);
      System.out.print(tamanio[i].nombre);
      long t1 = System.currentTimeMillis();
      pruebas[i].probar(a, reps);
```

```
      long t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
    }
  }
  public static void pruebaArray(int reps) {
    System.out.println("Probar array como lista");
    // En un array  sólo puede hacer las dos primeras pruebas:
    for(int i = 0; i < 2; i++) {
      String[] sa = new String[pruebas[i].tamanio];
      Arrays2.rellenarl(sa,
        Colecciones2.Paises.reset());
      List a = Arrays.asList(sa);
      System.out.print(pruebas[i].nombre);
      long t1 = System.currentTimeMillis();
      pruebas[i].probar(a, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // O, elegir el número de repeticiones
    // a través de la línea de comandos:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    System.out.println(reps + " repeticiones");
    pruebaArray(reps);
    prueba(new ArrayList(), reps);
    prueba(new LinkedList(), reps);
    prueba(new Vector(), reps);
  }
} ///:~
```

La clase interna **RealizarPrueba** es **abstracta**, para proporcionar una clase base para las pruebas específicas. Contiene un **String** que se imprime al empezar la prueba, un parámetro **tamanio** que se usará para contener la cantidad de elementos o repeticiones de prueba, un constructor para inicializar los campos, y un método **abstracto probar( )** que hace el trabajo. Todos los tipos de prueba se encuentran en el array **prueba**, que se inicializa con distintas clases internas anónimas heredadas de **RealizarPrueba**. Para añadir o eliminar probar, simplemente hay que añadir o eliminar una definición de clase interna del array, y todo ocurre automáticamente.

Para comparar el acceso a arrays con el acceso a contenedores (fundamentalmente contra **ArrayList**), se crea una prueba especial para arrays envolviéndolo como una **Lista** utilizando **Arrays.asList( )**. Fíjese que sólo se pueden hacer en esta clase las dos primeras pruebas, puesto que no se pueden insertar o eliminar elementos de un array.

La **Lista** que se pasa a **probar( )** se rellena primero con elementos, después se cronometran todos los pruebas del array **pruebas.** Los resultados variarán de una máquina a otra; se pretende que sólo den un orden de comparación de magnitudes entre el rendimiento de los distintos contenedores. He aquí un resultado de la ejecución:

Tipo	Obtener	Iteración	Insertar	Eliminar
array	1430	3850	No aplicable	No aplicable
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Como se esperaba, los arrays son más rápidos que cualquier contenedor si se persigue el acceso aleatorio e iteraciones. Se puede ver que los accesos aleatorios (**get( )**) son baratos para objetos **ArrayList** y caros en el caso de objetos **LinkedList**. (La iteración es *más rápida* para una **LinkedList** que para un **ArrayList,** lo que resulta bastante intuitivo). Por otro lado, las inserciones y eliminaciones que se lleven a cabo en el medio de la lista son drásticamente más rápidas en una **LinkedList** que en una **ArrayList** —*especialmente* las eliminaciones. **Vector** es generalmente menos rápido que **ArrayList**, por lo que se recomienda evitarlo; sólo está en la biblioteca para dar soporte al código antiguo (la única razón por la que funciona en este programa es por que fue adaptada para ser una **Lista** en Java 2). El mejor enfoque es probablemente elegir un **ArrayList** por defecto, y cambiar a **LinkedList** si se descubren problemas de rendimiento debido a muchas inserciones y eliminaciones en el centro de la lista. Y por supuesto, si se está trabajando con un grupo de elementos de tamaño fijo, debe usarse un array.

Elegir entre **Conjuntos**

Se puede elegir entre un objeto **TreeSet** y un objeto **HashSet**, dependiendo del tamaño del **Conjunto** (si se necesita producir una secuencia ordenada de un **Conjunto**, se usará un **TreeSet**). El siguiente programa de pruebas da una indicación de este equilibrio:

```
//: c09:RendimientoConjuntos.java
import java.util.*;
import com.bruceeckel.util.*;

public class RendimientoConjuntos {
  private abstract static class RealizarPrueba {
    String nombre;
    RealizarPrueba(String nombre) { this.nombre = nombre; }
    abstract void probar(Set s, int tamanio, int reps);
  }
  private static RealizarPrueba[] pruebas = {
```

```
    new RealizarPrueba("insertar") {
      void probar(Set s, int tamanio, int reps) {
        for(int i = 0; i < reps; i++) {
          s.clear();
          Colecciones2.rellenar(s,
            Colecctiones2.paises.inicializar(),tamanio);
        }
      }
    },
    new RealizarPrueba("contiene") {
      void probar(Set s, int tamanio, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < tamanio; j++)
            s.contains(Integer.toString(j));
      }
    },
    new RealizarPrueba("iteracion") {
      void probar(Set s, int tamanio, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = s.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void
  probar(Set s, int tamanio, int reps) {
    System.out.println("Probando " +
      s.getClass().getName() + " tamanio " + tamanio);
    Colecciones2.rellenarl(s,
      Colecciones2.paises.inicializar(), tamanio);
    for(int i = 0; i < pruebas.length; i++) {
      System.out.print(probar[i].nombre);
      long t1 = System.currentTimeMillis();
      pruebas[i].probar(s, tamanio, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)tamanio));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // O, elegir el número de repeticiones
    // a través de la línea de comandos:
```

```
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    // Pequeño:
    probar(new TreeSet(), 10, reps);
    probar(new HashSet(), 10, reps);
    // Medio:
    probar(new TreeSet(), 100, reps);
    probar(new HashSet(), 100, reps);
    // Grande:
    probar(new TreeSet(), 1000, reps);
    probar(new HashSet(), 1000, reps);
  }
} ///:~
```

Tipo	Tamaño prueba	Inserción	Contiene	Iteración
TreeSet	10	138,0	115,0	187,0
	100	189,5	151,1	206,5
	1000	150,6	177,4	40,04
HashSet	10	55,0	82,0	192,0
	100	45,6	90,0	202,2
	1000	36,14	106,5	39,39

La tabla siguiente muestra los resultados de la ejecución. (Por supuesto, éstos serán distintos en función del ordenador y la Máquina Virtual de Java que se esté usando; cada uno debería ejecutarlo):

El rendimiento de **HashSet** suele ser superior al de **TreeSet** en todas las operaciones (y especialmente en las dos operaciones más importantes: la inserción de elementos y la búsqueda). La única razón por la que existe **TreeSet** es porque mantiene sus elementos ordenados, por lo que se usa cuando se necesita un **Conjunto** ordenado.

Elegir entre **Mapas**

Al elegir entre implementaciones de **Mapas**, lo que más afecta al rendimiento es su tamaño; el siguiente programa de pruebas da una idea de este equilibrio:

```
//: c09:RendimientoMapas.java
// Demuestra diferencias de rendimiento en Mapas.
import java.util.*;
import com.bruceeckel.util.*;

public class RendimientoMapas {
```

```
private abstract static class RealizarPrueba {
  String nombre;
  RealizarPrueba(String nombre) { this.nombre = nombre; }
  abstract void probar(Map m, int tamanio, int reps);
}
private static RealizarPrueba[] pruebas = {
  new RealizarPrueba("poner") {
    void probar(Map m, int tamanio, int reps) {
      for(int i = 0; i < reps; i++) {
        m.clear();
        Colecciones2.rellenar(m,
          Colecciones2.geografia.inicializar(), tamanio);
      }
    }
  },
  new RealizarPrueba("quitar") {
    void probar(Map m, int tamanio, int reps) {
      for(int i = 0; i < reps; i++)
        for(int j = 0; j < tamanio; j++)
          m.get(Integer.toString(j));
    }
  },
  new RealizarPrueba("iteracion") {
    void probar(Map m, int tamanio, int reps) {
      for(int i = 0; i < reps * 10; i++) {
        Iterator it = m.entrySet().iterator();
        while(it.hasNext())
          it.next();
      }
    }
  },
};
public static void
probar(Map m, int tamanio, int reps) {
  System.out.println("Probando " +
    m.getClass().getName() + " tamanio " + tamanio);
  Colecciones2.rellenar(m,
    Colecciones2.geografia.inicializar(), size);
  for(int i = 0; i < pruebas.length; i++) {
    System.out.print(pruebas[i].nombre);
    long t1 = System.currentTimeMillis();
    pruebas[i].probar(m, tamanio, reps);
    long t2 = System.currentTimeMillis();
    System.out.println(": " +
      ((double)(t2 - t1)/(double)tamanio));
```

```
      }
    }
    public static void main(String[] args) {
      int reps = 50000;
      // O, elegir el número de repeticiones
      // a través de la línea de comandos:
      if(args.length > 0)
        reps = Integer.parseInt(args[0]);
      // pequeño:
      probar(new TreeMap(), 10, reps);
      probar(new HashMap(), 10, reps);
      probar(new Hashtable(), 10, reps);
      // Medio:
      probar(new TreeMap(), 100, reps);
      probar(new HashMap(), 100, reps);
      probar(new Hashtable(), 100, reps);
      // Grande:
      probar(new TreeMap(), 1000, reps);
      probar(new HashMap(), 1000, reps);
      probar(new Hashtable(), 1000, reps);
    }
} ///:~
```

Debido a que el factor es el tamaño del **mapa**, se verá que las pruebas de tiempos dividen el tiempo por el tamaño para normalizar cada medida. He aquí un conjunto de resultados. (Los de cada uno serán distintos.)

Tipo	Tamaño de la prueba	Poner	Quitar	Iteración
TreeMap	10	143,0	110,0	186,0
	100	201,1	188,4	280,1
	1000	222,8	205,2	40,7
HashMap	10	66,0	83,0	197,0
	100	80,7	135,7	278,5
	1000	48,2	105,7	41,4
HashTable	10	61,0	93,0	302,0
	100	90,6	143,3	329,0
	1000	54,1	110,95	47,3

Como se podía esperar, el rendimiento de **Hashtable** es equivalente al de **HashMap**. (También se puede ver que **HashMap** es generalmente un poco más rápida. Se pretende que **HashMap** reemplace a **Hashtable**.) El **TreeMap** es generalmente más lento que el **HashMap**, así que ¿por qué usarlo? Así se podría usar en vez de como un **Mapa**, como una forma de crear una lista ordenada. El comportamiento de un árbol es tal que siempre está en orden y no hay que ordenarlo de forma especial. Una vez que se rellena un **TreeMap** se puede invocar a **keySet( )** para conseguir una vista del **Conjunto** de las claves, después a **toArray( )** para producir un array de esas claves. Se puede usar el método **estático Arrays.binarySearch( )** (que se discutirá más tarde) para encontrar rápidamente objetos en el array ordenado. Por supuesto, sólo se haría esto si, por alguna razón, el comportamiento de **HashMap** fuera inaceptable, puesto que **HashMap** está diseñado para encontrar elementos rápidamente. También se puede crear fácilmente un **HashMap** a partir de un **TreeMap** con una única creación de objetos. Finalmente, cuando se usa un **Mapa** la primera elección debe ser **HashMap**, y sólo si se necesita un **Mapa** constantemente ordenado deberá usarse **TreeMap**.

Ordenar y buscar elementos en **Listas**

Las utilidades para llevar a cabo la ordenación y búsqueda de elementos en **Listas** tienen los mismos nombres y parámetros que las de ordenar arrays de objetos, pero son métodos **estáticos** de **Colecciones** en vez de **Arrays**. Aquí hay un ejemplo, modificado a partir de **BuscarEnArray.java**:

```
//: c09:OrdenarBuscarEnLista.java
// Ordenando y buscando Listas con 'Colecciones.'
import com.bruceeckel.util.*;
import java.util.*;

public class OrdenarBuscarEnLista {
  public static void main(String[] args) {
    List lista = new ArrayList();
    Colecciones2.rellenar(list,
      Colecciones2.capitales, 25);
    System.out.println(lista + "\n");
    Collections.shuffle(lista);
    System.out.println("Despues de desordenar: "+lista);
    Collections.sort(lista);
    System.out.println(lista + "\n");
    Object clave = lista.get(12);
    int indice =
      Collections.binarySearch(lista, clave);
    System.out.println("Posicion de " + clave +
      " es " + indice + ", lista.get(" +
      indice + ") = " + lista.get(indice));
    ComparadorAlfabetico comp =
      new ComparadorAlfabetico();
    Colecciones.sort(lista, comp);
```

```
        System.out.println(lista + "\n");
        clave = lista.get(12);
        indice =
          Collections.binarySearch(lista, clave, comp);
        System.out.println("Posicion de " + key +
          " es " + indice + ", lista.get(" +
          indice + ") = " + lista.get(indice));
    }
} ///:~
```

El uso de estos métodos es idéntico al de los **Arrays**, pero se está usando una **Lista** en vez de un array. Al igual que ocurre al buscar y ordenar en arrays, si se ordena usando un **Comparador**, hay que usar el **binarySearch( )** del propio **Comparador**.

Este programa también muestra el método **shuffle( )** de las **Colecciones**, que genera un orden aleatorio para una **Lista**.

Utilidades

Hay otras muchas utilidades en las clases de tipo **Colección**:

enumeration(Collection)	Devuelve una **Enumeración** de las antiguas para el parámetro.
max(Collection) **min(Collection)**	Devuelve el elemento máximo o mínimo del parámetro utilizando el método natural de comparación para los objetos de la **Colección**.
max(Collection, Comparator) **min(Collection, Comparator)**	Devuelve el elemento máximo o mínimo de la **Colección** utilizando el **Comparador**.
reverse( )	Invierte el orden de todos los elementos.
copy(List destino, List origen)	Copia los elementos de origen a destino.
fill(List lista, Object o)	Reemplaza todos los elementos de la lista con o.
nCopies(int n, Object o)	Devuelve una **Lista** inmutable de tamaño n cuyas referencias apuntan a o.

Fíjese que **min( )** y **max( )** trabajan con objetos **Colección**, en vez de **Listas**, por lo que no hay que preocuparse de si la **Colección** está o no ordenada. (Como se mencionó anteriormente, *hay* que ordenar una **Lista** utilizando el método **sort( )** o un array antes de llevar a cabo una **binarySearch ( )**.)

Hacer inmodificable una **Colección** o un **Mapa**

A menudo es conveniente crear una versión de sólo lectura de una **Colección** o un **Mapa**. La clase de tipo **Colección** permite hacer esto pasando el contenedor original a un método que devuelve una versión de sólo lectura. Hay cuatro variantes de este método, una para **Collection** (por si no se desea crear una **Colección** de un tipo más específico), **List**, **Set** y **Map**. Este ejemplo muestra la forma adecuada de construir versiones de sólo lectura de cada uno de ellos:

```
//: c09:SoloLectura.java
// Usando los métodos Collections.unmodifiable.
import java.util.*;
import com.bruceeckel.util.*;

public class SoloLectura{
  static Colecciones2.StringGenerador gen =
    Colecciones2.Paises;
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Colecciones2.rellenar(c, gen, 25); // Insertar datos
    c = Collections.unmodifiableCollection(c);
    System.out.println(c); // La lectura es correcta
    c.add("uno"); // No se puede cambiar

    List a = new ArrayList();
    Colecciones2.rellenar(a, gen.reset(), 25);
    a = Collections.unmodifiableList(a);
    ListIterator lit = a.listIterator();
    System.out.println(lit.next()); // La lectura es correcta
    lit.add("uno"); // No se puede cambiar

    Set s = new HashSet();
    Colecciones2.rellenar(s, gen.inicializar(), 25);
    s = Collections.unmodifiableSet(s);
    System.out.println(s); // La lectura es correcta
    //! s.add("uno"); // No se puede cambiar

    Map m = new HashMap();
    Colecciones2.rellenar(m,
      Colecciones2.geografia, 25);
    m = Collections.unmodifiableMap(m);
    System.out.println(m); // La lectura es correcta
    //! m.put("Javier", "Hola!");
  }
} ///:~
```

En cada caso, hay que rellenar el contenedor con datos significativos *antes* de hacerlos de sólo lectura. Una vez cargado, el mejor enfoque es reemplazar la referencia existente por la producida por la llamada "inmodificable". De esa forma, no se corre el riesgo de cambiar accidentalmente los contenidos una vez convertida en inmodificable. Por otro lado, esta herramienta también permite mantener un contenedor modificado **privado** dentro de una clase, y devolver una referencia de sólo lectura a ese contenedor desde una llamada a un método. Por tanto se puede cambiar dentro de la clase, y el resto de gente simplemente puede leerlo.

Llamar al método "inmodificable" para un tipo particular no provoca comprobación en tiempo de ejecución, pero una vez que se ha dado la transformación, todas las llamadas a métodos que modifiquen los contenidos de un contenedor en particular producirán una excepción de tipo **UnsupportedOperationException**.

Sincronizar una **Colección** o **Mapa**

La palabra clave **synchronized** es una parte importante del *multihilo*, un tema aún más complicado en el que no se entrará hasta el Capítulo 14. Aquí, simplemente se destaca que la clase **Collections** contiene una forma de sincronizar automáticamente un contenedor entero. La sintaxis es similar a los métodos "inmodificable":

```
//: c09:Sincronizacion.java
// Uso de los métodos Collections.synchronized.
import java.util.*;

public class Sincronizacion {
  public static void main(String[] args) {
    Collection c =
      Collections.synchronizedCollection(
        new ArrayList());
    List list = Collections.synchronizedList(
      new ArrayList());
    Set s = Collections.synchronizedSet(
      new HashSet());
    Map m = Collections.synchronizedMap(
      new HashMap());
  }
} ///:~
```

En este caso, se pasa inmediatamente el contenedor nuevo a través del método "sincronizado" apropiado; de esa manera no hay forma de exponer accidentalmente la versión sin sincronizar.

Fallo rápido

Los contenedores de Java tienen también un mecanismo para evitar que los contenidos de un contenedor sean modificados por más de un proceso. El problema se da cuando se está iterando a través de un contenedor y algún proceso irrumpe insertando, retirando o cambiando algún objeto de

ese contenedor. Ese objeto puede estar aún por procesar o ya se ha pasado por él, o incluso puede ser que se den problemas al invocar a **size( )** —hay demasiados escenarios que conducen al desastre. La biblioteca de contenedores de Java incorpora un mecanismo de *fallo rápido* que vigila que no se den en el contenedor más cambios que aquéllos de los que cada proceso se responsabiliza. Si detecta que alguien más está modificando el contenedor, produce inmediatamente una excepción **ConcurrentModificationException**. Éste es el aspecto *"Fallo rápido"* —no intenta detectar un problema más tarde utilizando algoritmos más complejos.

Es bastante sencillo ver el mecanismo fallo rápido en operación —todo lo que se tiene que hacer es crear un iterador y añadir algo a la colección apuntada por el iterador, como en:

```
//: c09:FalloRapido.java
// Demuestra el comportamiento "fallo rápido".
import java.util.*;

public class FalloRapido {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Iterator it = c.iterator();
    c.add("Un objeto");
    // Origina una excepción:
    String s = (String)it.next();
  }
} ///:~
```

Se da una excepción porque se ha colocado algo en el contenedor *después* de que se adquiere un iterador para el contenedor. La posibilidad de que dos partes del programa puedan estar modificando el mismo contenedor produce un estado incierto, por lo que la excepción notifica que habría que cambiar el código —en este caso, hay que adquirir el iterador *después* de que se hayan añadido todos los elementos del contenedor.

Fíjese que nadie se puede beneficiar de este tipo de monitorización cuando se esté accediendo a los elementos de una **Lista** utilizando **get( )**.

Operaciones no soportadas

Es posible convertir un array en una **Lista** con el método **Arrays.asList( )**:

```
//: c09:NoSoportable.java
// ¡En ocasiones los métodos definidos en las
// interfaces Collection no funcionan!
import java.util.*;

public class NoSoportable {
  private static String[] s = {
    "uno", "dos", "tres", "cuatro", "cinco",
```

```
      "siete", "ocho", "nueve", "diez",
   };
   static List a = Arrays.asList(s);
   static List a2 = a.subList(3, 6);
   public static void main(String[] args) {
      System.out.println(a);
      System.out.println(a2);
      System.out.println(
         "a.contains(" + s[0] + ") = " +
         a.contains(s[0]));
      System.out.println(
         "a.containsAll(a2) = " +
         a.containsAll(a2));
      System.out.println("a.isEmpty() = " +
         a.isEmpty());
      System.out.println(
         "a.indexOf(" + s[5] + ") = " +
         a.indexOf(s[5]));
      // Recorrer hacia atrás:
      ListIterator lit = a.listIterator(a.size());
      while(lit.hasPrevious())
         System.out.print(lit.previous() + " ");
      System.out.println();
      // Poner distintos elementos a los valores:
      for(int i = 0; i < a.size(); i++)
         a.set(i, "47");
      System.out.println(a);
      // Compila, pero no se ejecuta:
      lit.add("X"); // Operación no soportada
      a.clear(); // No soportada
      a.add("once"); // No soportada
      a.addAll(a2); // No soportada
      a.retainAll(a2); // No soportada
      a.remove(s[0]); // No soportada
      a.removeAll(a2); // No soportada
   }
} ///:~
```

Se descubrirá que sólo están implementados algunas de las interfaces de **List** y **Collection**. El resto de los métodos causan la aparición no bienvenida de algo denominado **Unsupported OperationException**. Se aprenderá todo lo relativo a las excepciones en el capítulo siguiente, pero en resumidas cuentas, la **interfaz Collection** —al igual que algunos de las demás interfaces de la biblioteca de contenedores de Java— contienen métodos "opcionales", que podrían estar o no "soportados" en la clase concreta que **implementa** esa **interfaz**. Llamar a un método no

soportado causa una **UnsupportedOperationException** para indicar un error de programación.

"¡Qué!" dice uno incrédulo. "¡La razón principal de las **interfaces** y las clases base es que prometen que estos métodos harán algo significativo! Esto rompe esa promesa —dice que los métodos invocados *no* sólo *no* desempeñarán un comportamiento significativo, sino que pararán el programa! ¡Nos acabamos de cargar la seguridad a nivel de tipos!"

No es para tanto. Con un objeto de tipo **Collection**, **Set**, **List** o **Map**, el compilador sigue restringiendo de forma que sólo se invoquen los métodos de esta **interfaz**, a diferencia de lo que ocurre en Smalltalk (en el que se puede invocar a cualquier método para cualquier objeto, no descubriendo hasta tiempo de ejecución que la llamada no hace nada). Además, la mayoría de los métodos que toman una **Colección** como un parámetro sólo leen de la misma —y *ninguno* de los métodos de "lectura" de las **Colecciones** es opcional.

Este enfoque evita una explosión de interfaces en el diseño. Otros diseños para las bibliotecas de contenedores siempre parecen acabar con una plétora de interfaces para describir cada una de las variaciones, siendo por consiguiente, difíciles de aprender. Ni siquiera es posible capturar todos los casos especiales en **interfaces**, porque siempre habrá alguien capaz de inventar una **interfaz** nueva. El enfoque de "operación no soportada" logra una meta importante de la biblioteca de contenedores de Java: los contenedores se vuelven fáciles de aprender y usar; las operaciones no soportadas son un caso especial que puede aprenderse más adelante. Sin embargo, para que este enfoque funcione:

1. La **UnsupportedOperationException** debe ser un evento extraño. Es decir, en la mayoría de clases deberían funcionar todas las operaciones, y sólo en casos especiales podría haber una operación sin soporte. Esto es cierto en la biblioteca de contenedores de Java, dado que las clases que se usan el 99 % de las veces —**ArrayList**, **LinkedList**, **HashSet** y **HashMap**, al igual que las otras implementaciones concretas— soportan todas las operaciones. El diseño sí que proporciona una "puerta trasera" si se desea crear una nueva **Colección** sin proporcionar definiciones significativas para todos los métodos de la Interfaz **Collection**, haciendo, sin embargo, que encaje en la biblioteca existente.

2. Cuando una operación *no* tiene soporte, debería haber una probabilidad razonable de que aparezca una **UnsupportedOperationException** en tiempo de implementación, más que cuando se haya entregado el producto al cliente. Después de todo, indica un error de programación: se ha usado una implementación de forma incorrecta. Este punto es menos detectable, y es donde se pone en juego la naturaleza experimental del diseño. Sólo el tiempo permitirá ir averiguando con certeza cómo funciona.

En el ejemplo de arriba, **Arrays.asList( )** produce una **Lista** soportada por un array de tamaño fijo. Por consiguiente, tiene sentido que sólo sean las operaciones soportadas las que no cambien el tamaño del array. Si, por otro lado, se requiriera una nueva **interfaz** para expresar este tipo de comportamiento distinto (denominado, quizás, "**FixedSizeList**"), conduciría directamente a la complejidad y pronto se dejaría de saber dónde empezar a intentar usar la biblioteca.

La documentación para un método que tome una **Collection**, **List**, **Set** o **Map** como parámetro debería especificar cuál de los métodos opcionales debe implementarse. Por ejemplo, la ordenación requiere métodos **set( )** e **Iterator.set( )**, pero no **add( )** y **remove( )**.

Contenedores de Java 1.0/1.1

Desgraciadamente, se ha escrito mucho código utilizando los contenedores de Java 1.0/1.1, e incluso se escribe código nuevo utilizando estas clases. Por tanto, aunque nunca se debería utilizar nuevo código utilizando los contenedores viejos, hay que ser consciente de que existen. Sin embargo, los contenedores viejos eran bastante limitados, por lo que no hay mucho que decir de los mismos. (Puesto que son cosas del pasado, intentaremos evitar poner demasiado énfasis en algunas horribles decisiones de diseño.)

Vector y enumeration

La única secuencia cuyo tamaño podía autoexpandirse en Java 1.0/1.1 era el **Vector**, y por tanto se usó mucho. Tiene demasiados defectos como para describirlos aquí (véase la primera edición de este libro, disponible en el CD ROM de este libro y como descarga gratuita de: *http://www.Bruce Eckel.com*). Básicamente, se puede pensar que es un **ArrayList** con nombres de métodos largos y extraños. En la biblioteca de contenedores de Java 2 se ha adaptado **Vector,** de forma que pudiera encajar como una **Colección** y una **Lista**, por lo que en el ejemplo siguiente, el método **Colecciones2.rellenar( )** se usa exitosamente. Esto resulta un poco extraño, pues puede confundir a la gente que puede llegar a pensar que **Vector** ha mejorado, cuando, de hecho, sólo se ha incluido para soportar el código previo a Java 2.

La versión Java 1.0/1.1 del iterador decidió inventar un nuevo nombre: "enumeración", en vez de utilizar un término con el que todo el mundo ya era familiar. La interfaz **Enumeration** es más pequeño que **Iterator**, con sólo dos métodos, y usa nombres de método más largos: **boolean hasMoreElements( )** devuelve **verdadero** si esta enumeración contiene más elementos, y **Object nextElement( )** devuelve el siguiente elemento de la enumeración actual si es que hay alguno (de otra manera, produce una excepción).

Enumeration es sólo una interfaz, no una implementación, e incluso las bibliotecas nuevas siguen usando la vieja **Enumeration** —que es desdichada, pero generalmente inocua. Incluso aunque se debería usar siempre **Iterador** cuando se pueda, hay que estar preparados para bibliotecas que quieran hacer uso de una **Enumeración.**

Además, se puede producir una **Enumeración** para cualquier **Colección** utilizando el método **Collections.enumeration( )**, como se ve en este ejemplo:

```
//: c09:Enumeraciones.java
// Java 1.0/1.1 Vector y Enumeration.
import java.util.*;
import com.bruceeckel.util.*;
```

```
class Enumeraciones {
  public static void main(String[] args) {
    Vector v = new Vector();
    Colecciones2.rellenar(
      v, Collecciones2.paises, 100);
    Enumeration e = v.elements();
    while(e.hasMoreElements())
      System.out.println(e.nextElement());
    // devuelve una enumeración de una colección:
    e = Collections.enumeration(new ArrayList());
  }
} ///:~
```

El **Vector** de Java 1.0/1.1 sólo tiene un método **addElement( )**, pero **rellenar( )** hace uso del método **add( )** que se adaptó cuando se convirtió **Vector** en **Lista**. Para producir una **Enumeración**, se invoca a **elements( )**, por lo que se puede usar para llevar a cabo una iteración hacia adelante.

La última línea crea un **ArrayList** y usa **enumeration( )** para adaptar una **Enumeración** del **iterador** de **ArrayList**. Por consiguiente, si se tiene código viejo que requiera una **Enumeración**, se pueden seguir usando los nuevos contenedores.

Hashtable

Como se ha visto en la comparación de rendimientos en este capítulo, la **Hashtable** básica es muy similar al **HashMap**, incluso en los nombres de método. No hay razón para usar **Hashtable** en vez de **HashMap** en el código nuevo.

Pila (Stack)

El concepto de pila ya se presentó anteriormente, con la **LinkedList**. Lo extraño de la **Pila** de Java 1.0/1.1 es que en vez de usar **Vector** como bloque constructivo, la **Pila** se *hereda* de **Vector**. Por tanto tiene todas las características y comportamientos de un **Vector** más algunos comportamientos propios de **Pila**. Es difícil saber si los diseñadores decidieron explícitamente que ésta fuera una forma especialmente útil de hacer las cosas, o si fue simplemente un diseño ingenuo.

He aquí una simple demostración de una **Pila** que introduce cada línea de un array de **Cadenas de caracteres**:

```
//: c09:Pilas.java
// Demostración de la clase Stack.
import java.util.*;

public class Pilas {
  static String[] meses = {
    "Enero", "Febrero", "Marzo", "Abril",
```

```
      "Mayo", "Junio", "Julio", "Agosto", "Septiembre",
      "Octubre", "Noviembre", "Diciembre" };
   public static void main(String[] args) {
      Stack pila = new Stack();
      for(int i = 0; i < meses.length; i++)
         pila.push(meses[i] + " ");
      System.out.println("pila = " + pila);
      // Tratando una pila como un Vector:
      pila.addElement("La ultima linea");
      System.out.println(
         "elemento 5 = " + pila.elementAt(5));
      System.out.println("sacando elementos:");
      while(!pila.empty())
         System.out.println(pila.pop());
   }
} ///:~
```

Cada línea del array **meses** se inserta en la **Pila** con **push( )** y posteriormente se la toma de la cima de la pila con **pop( )**. Todas las operaciones **Vector** también se ejecutan en el objeto **Stack**. Esto es posible porque, gracias a la herencia, un objeto de tipo **Stack** *es* un **Vector**. Por consiguiente, todas las operaciones que puedan llevarse a cabo sobre un **Vector** también pueden ejecutarse en una **Pila** como **elementAt( )**.

Como se mencionó anteriormente, se debería usar un objeto **LinkedList** cuando se desee comportamiento de pila.

Conjunto de bits (BitSet)

Un **Conjunto de bits** se usa si se desea almacenar eficientemente gran cantidad de información. Es eficiente sólo desde el punto de vista del tamaño; si se busca acceso eficiente, es ligeramente más lento que usar un array de algún tipo nativo.

Además, el tamaño mínimo de **Conjunto de bits** es el de un **long**: 64 bits. Esto implica que si se está almacenando algo menor, como 8 bits, un **Conjunto de bits** sería un derroche; es mejor crear una clase o simplemente un array, para guardar indicadores cuando el tamaño es importante.

Un contenedor normal se expande al añadir más elementos, y un **Conjunto de bits** también. El ejemplo siguiente muestra el funcionamiento del **Conjunto de bits**:.

```
//: c09:Bits.java
// Demostración de BitSet.
import java.util.*;

public class Bits {
   static void escribirBitset(BitSet b) {
      System.out.println("bits: " + b);
```

```
    String bbits = new String();
    for(int j = 0; j < b.size() ; j++)
      bbits += (b.get(j) ? "1" : "0");
    System.out.println("patron de bit: " + bbits);
  }
  public static void main(String[] args) {
    Random aleatorio = new Random();
    // Toma el LSB de nextInt():
    byte bt = (byte)aleatorio.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >=0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    System.out.println("valor byte: " + bt);
    escribirBitSet(bb);

    short st = (short)aleatorio.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >=0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    System.out.println("valor short: " + st);
    escribirBitSet(bs);

    int it = aleatorio.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >=0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    System.out.println("valor int: " + it);
    escribirBitSet(bi);

    // Prueba conjunto de bits >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    System.out.println("poner a uno el bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("poner a uno el bit 255: " + b255);
```

```
        BitSet b1023 = new BitSet(512);
        b1023.set(1023);
        b1023.set(1024);
        System.out.println("poner a uno el bit 1023: " + b1023);
    }
} ///:~
```

El generador de números aleatorios se usa para crear un **byte**, **short**, e **int** al azar, transformando cada uno en el patrón de bits correspondiente en el **Conjunto de bits**. Esto funciona bien porque un **Conjunto de bits** tiene 64 bits, por lo que ninguno de éstos provoca un incremento de tamaño. Posteriormente se crea un **Conjunto de bits** de 512 bits. El constructor asigna espacio de almacenamiento para el doble de bits. Sin embargo, se sigue pudiendo poner a uno el bit 1.024 o mayor.

Resumen

Para repasar los contenedores proporcionados por la biblioteca estándar de Java:

1. Un array asocia índices numéricos a objetos. Guarda objetos de un tipo conocido por lo que no hay que convertir el resultado cuando se está buscando un objeto. Puede ser multidimensional, y puede guardar datos primitivos. Sin embargo, no se puede cambiar su tamaño una vez creado.

2. Una **Colección** guarda elementos sencillos, mientras que un **Mapa** guarda pares asociados.

3. Como un array, una **Lista** también asocia índices numéricos a los objetos —se podría pensar que los arrays y **Listas** son contenedores ordenados. La **Lista** se redimensiona automáticamente al añadir más elementos. Pero una **Lista** sólo puede guardar **referencias a Objetos**, por lo que no guardará datos primitivos, y siempre hay que convertir el resultado para extraer una referencia a un **Objeto** fuera del contenedor.

4. Utilice un **ArrayList** si se están haciendo muchos accesos al azar, y una **LinkedList** si se están haciendo muchas inserciones y eliminaciones en el medio de la lista.

5. El comportamiento de las colas, bicolas y pilas se proporciona mediante **LinkedList**.

6. Un **Mapa** es una forma de asociar valores no números, sino *objetos* con otros objetos. El diseño de un **HashMap** se centra en el acceso rápido, mientras que un **TreeMap** mantiene sus claves ordenadas, no siendo, por tanto, tan rápido como un **HashMap**.

7. Un **Conjunto** sólo acepta uno de cada tipo de objeto. Los **HashSets** proporcionan búsquedas extremadamente rápidas, mientras que los **TreeSets** mantienen los elementos ordenados.

8. No hay necesidad de usar las clases antiguas **Vector**, **Hashtable** y **Stack** en el código nuevo.

Los contenedores son herramientas que se pueden usar en el día a día para construir programas más simples, más potentes y más efectivos.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com.*

1. Crear un array de **elementos de tipo doble** y rellenarlo haciendo uso de **GeneradorDobleAleatorio**. Imprimir los resultados.

2. Crear una nueva clase denominada **Jerbo** con un **atributo numeroJerbos de tipo entero** que se inicialice en el constructor (similar al ejemplo **Ratón** de este capítulo). Darle un método denominado **saltar( )** que imprima qué numero de jerbo se está saltando. Crear un **ArrayList** y añadir un conjunto de objetos **Jerbo** a la **Lista**. Ahora usar el método **get( )** para recorrer la **Lista** y llamar a **saltar( )** para cada **Jerbo**.

3. Modificar el Ejercicio 2 de forma que se use un **Iterator** para recorrer la **Lista** mientras se llama a **saltar( )**.

4. Coger la clase **Jerbo** del Ejercicio 2 y ponerla en un **Mapa** en su lugar, asociando el nombre del **Jerbo** como un **String** (la clave) por cada **Jerbo** (el valor) que se introduzca en la tabla. Conseguir un **Iterador** para el **keySet( )** y utilizarlo para recorrer **Mapa**, buscando el **Jerbo** para cada clave e imprimiendo la clave y ordenando al **Jerbo** que **saltar( )**.

5. Crear una **Lista** (intentarlo tanto con **ArrayList** como con **LinkedList**) y rellenarla con **Colecciones2.paises**. Ordenar la lista e imprimirla, después aplicar **Collections.shuffle( )** a la lista repetidamente, imprimiéndola cada vez para ver como el método **shuffle( )** genera una lista aleatoriamente distinta cada vez.

6. Demostrar que no se puede añadir nada que no sea un **Ratón** a una **ListaRaton**.

7. Modificar **ListaRaton.java** de forma que herede de **ArrayList** en vez de hacer uso de la composición. Demostrar el problema con este enfoque.

8. Modificar **GatosYPerros.java** creando un contenedor **Gatos** (utilizando **ArrayList**) que sólo aceptará y retirará objetos **Gato**.

9. Crear un contenedor que encapsule un array de cadena de caracteres, y que sólo añada y extraiga cadena de caracteres, de forma que no haya problemas de conversiones durante su uso. Si el array interno no es lo suficientemente grande para la siguiente adición, redimensionar automáticamente el contenedor. En el método **main( )** comparar el rendimiento del contenedor con un **ArrayList** que almacene cadena de caracteres.

10. Repetir el Ejercicio 9 para un contenedor de **enteros**, y comparar el rendimiento con el de un **ArrayList** que guarde objetos **Integer**. En la comparación de rendimiento, incluir el proceso de incrementar cada objeto del contenedor.

11. Utilizando las utilidades de **com.bruceeckel.util**, crear un array de cada tipo primitivo y de **Cadena de Caracteres**, rellenar después cada array utilizando un generador apropiado, e imprimir cada array usando el método **escribir( )** adecuado.

12. Crear un generador que produzca nombres de personajes de alguna película (se puede usar por defecto *Blanca Nieves* o *La Guerra de las galaxias*), y que vuelva a comenzar por el principio al quedarse sin nombres. Utilizar las utilidades de **com.bruceeckel.util** para rellenar un array, un **ArrayList**, una **LinkedList** y ambos tipos de **Set**, para finalmente imprimir cada contenedor.

13. Crear una clase que contenga dos objetos **String**, y hacerla **Comparable,** de forma que la comparación sólo se encargue del primer **String**. Rellenar un array y un **ArrayList** con objetos de la clase, utilizando el generador **geografía**. Demostrar que la ordenación funciona correctamente. Hacer ahora un **Comparador** que sólo se encargue del segundo **String** y demostrar que la ordenación funciona correctamente; llevar a cabo también una búsqueda binaria haciendo uso del **Comparador**.

14. Modificar el Ejercicio 13 de forma que se siga un orden alfabético.

15. Utilizar **Arrays2.GeneradorStringAleatorio** para rellenar un **TreeSet** pero usando ordenación alfabética. Imprimir el **TreeSet** para verificar el orden.

16. Crear un **ArrayLit** y una **LinkedList**, y rellenar cada uno utilizando el generador **Collections2.capitales**. Imprimir cada lista utilizando un **Iterador** ordinario, y después insertar una lista dentro de la otra utilizando un **ListIterator**, intercalándolas. Llevar a cabo ahora la inserción empezando al final de la primera lista y recorriéndola hacia atrás.

17. Escribir un método que use un **Iterador** para recorrer una **Colección** e imprimir el **hashCode( )** de cada objeto del contenedor. Rellenar todos los tipos distintos de **Colección** con objetos y aplicar el método a cada contenedor.

18. Reparar el problema de **RecursividadInfinita.java**.

19. Crear una clase, después hacer un array inicializado de objetos de esa clase. Rellenar una **Lista** con ese array. Crear un subconjunto de la **Lista** utilizando **subList( )**, y eliminar después este subconjunto de la **Lista** utilizando **removeAll( )**.

20. Cambiar el Ejercicio 6 del Capítulo 7, de forma que use un **ArrayList** para guardar los **Roedores** y un **Iterador** para recorrer la secuencia de objetos **Roedores**. Recordar que un **ArrayList** guarda sólo **Objetos** por lo que hay que hacer una conversión al acceder a los objetos **Roedor** individuales.

21. Siguiendo el ejemplo **Cola.java**, crear una clase **Bicola** y probarla.

22. Utilizar un **TreeMap** en **Estadisticas.java**. Añadir ahora código que pruebe la diferencia de rendimiento entre **HashMap** y **TreeMap** en el programa.

23. Producir un **Mapa** y un **Conjunto** que contengan todos los países que empiecen por "A".

24. Utilizando **Colecciones2.paises**, rellenar un **Set** varias veces con los mismos datos y verificar que el **Set** acaba con sólo una instancia de cada. Intentarlo con los dos tipos de **Set**.

25. A partir de **Estadisticas.java**, crear un programa que ejecute la prueba repetidamente y comprueba si alguno de los números tiende a aparecer en los resultados más a menudo que otros.

26. Volver a escribir **Estadisticas.java** utilizando un **HashSet** de objetos **Contador** (habrá que modificar **Contador** de forma que trabaje en el **HashSet**). ¿Qué enfoque parece mejor?

27. Modificar la clase del Ejercicio 13 de forma que trabaje con objetos **HashSet**, y utilizar una clave de objeto **HashMap**.

28. Utilizando como inspiración **MapaLento.java**, crear un **MapaLento**.

29. Aplicar las pruebas de **Mapas1.java** a **MapaLento** para verificar que funciona. Arreglar todo lo que no funcione correctamente en **MapaLento**.

30. Implementar el resto de la interfaz **Map** para **MapaLento**.

31. Modificar **RendimientoMapa.java** para que incluya tests de **MapaLento**.

32. Modificar **MapaLento** para que en vez de objetos **ArrayList** guarde un único **ArrayList** de objetos **MPar**. Verificar que la versión modificada funcione correctamente. Utilizando **RendimientoMapa.java**, probar la velocidad del nuevo **Mapa**. Ahora cambiar el método **put( )** de forma que haga un **sort( )** después de que se introduzca cada par, y modificar **get( )** para que use **Collections.binarySearch( )** para buscar la clave. Comparar el rendimiento de la nueva versión con el de la vieja.

33. Añadir un campo **de tipo carácter** a **CuentaString** que se inicialice también en el constructor, y modificar los métodos **HashCode( )** y **equals( )** para incluir el valor de este **de tipo carácter**.

34. Modificar **SimpleHashMap** de forma que informe sobre colisiones, y probarlo añadiendo el mismo conjunto de datos dos veces, de forma que se observen colisiones.

35. Modificar **SimpleHashMap** de forma que informe del número de "intentos" necesarios cuando se dan colisiones. Es decir, ¿cuántas llamadas a **next( )** hay que hacer en los **Iteradores** que recorren las **LinkedLists** para encontrar coincidencias?

36. Implementar los métodos **clear( )** y **remove( )** para **SimpleHashMap**.

37. Implementar el resto de la interfaz **Map** para **SimpleHashMap**.

38. Añadir un método **privado rehash( )** para **SimpleHashMap** al que se invoca cuando el factor de carga excede 0,75. Durante el rehashing doblar el número de posiciones, después buscar el primer número primo mayor para determinar el nuevo número de posiciones.

39. Siguiendo el ejemplo de **SimpleHashMap.java**, crear y probar un **SimpleHashSet**.

40. Modificar **SimpleHashMap** para que use **ArrayList** en vez de **LinkedList**. Modificar **RendimientoMapa.java** para comparar el rendimiento de ambas implementaciones.

41. Utilizando la documentación HTML del JDK (descargable de *http://java.sun.com*), buscar la clase **HashMap**. Crear un **HashMap**, rellenarlo con elementos y determinar el factor de carga. Probar la velocidad de búsqueda con este mapa, y después intentar incrementar la velocidad haciendo un nuevo **HashMap** con una capacidad inicial más grande y copiando el mapa viejo en el nuevo, ejecutando de nuevo la prueba de velocidad de búsqueda en el nuevo mapa.

42. En el Capítulo 8, localizar el ejemplo **ControlesInvernadero.java**, que consta de tres ficheros. En **Controlador.java**, la clase **ConjuntoEventos** es simplemente un contenedor. Cambiar el código para usar una **LinkedList** en vez de un **ConjuntoEventos**. Esto exigirá más que simplemente reemplazar **ConjuntoEventos** con **LinkedList**; también se necesitará usar un **Iterador** para recorrer el conjunto de eventos.

43. (Desafío). Escribir una clase mapa propia con hashing, personalizada para un tipo de clave particular: **String** en este caso. No heredarla de **Map**. En su lugar, duplicar los métodos de forma que los métodos **put( )** y **get( )** tomen específicamente objetos **String**, en vez de **Objects**, como claves. Todo lo relacionado con las claves no debería usar tipos genéricos, sino que debería funcionar con **Strings**, para evitar el coste de las conversiones hacia arriba y hacia abajo. La meta es hacer la implementación general más rápida posible. Modificar **RendimientoMapa.java** de forma que pruebe esta implementación contra un **HashMap**.

44. (Desafío). Encontrar el código fuente de **List** en la biblioteca de código fuente de Java que viene con todas las distribuciones de Java. Copiar este código y hacer una versión especial llamada **ListaEnteros** que guarde sólo **números enteros**. Considerar qué implicaría hacer una versión especial de **List** para todos los tipos primitivos. Considerar ahora qué ocurre si se desea hacer una clase lista enlazada que funcione con todos los tipos primitivos. Si alguna vez se llegaran a implementar tipos parametrizados en Java, proporcionarán la forma de hacer este trabajo automáticamente (además de muchos otros beneficios).

10: Manejo de errores con excepciones

La filosofía básica de Java es que "el código mal formado no se ejecutará".

El momento ideal para capturar un error es en tiempo de compilación, antes incluso de intentar ejecutar el programa. Sin embargo, no todos los errores se pueden detectar en tiempo de compilación. El resto de los problemas deberán ser manejados en tiempo de ejecución, mediante alguna formalidad que permita a la fuente del error pasar la información adecuada a un receptor que sabrá hacerse cargo de la dificultad de manera adecuada.

En C y en otros lenguajes anteriores, podía haber varias de estas formalidades, que generalmente se establecían por convención y no como parte del lenguaje de programación. Habitualmente, se devolvía un valor especial o se ponía un indicador a uno, y el receptor se suponía, que tras echar un vistazo al valor o al indicador, determinaría la existencia de algún problema. Sin embargo, con el paso de los años, se descubrió que los programadores que hacían uso de bibliotecas tendían a pensar que eran invencibles —como en "Sí, puede que los demás cometan errores, pero no en *mi* código". Por tanto, y lógicamente, éstos no comprobaban que se dieran condiciones de error (y en ocasiones las condiciones de error eran demasiado estúpidas como para comprobarlas)[1]. Si uno *fuera* tan exacto como para comprobar todos los posibles errores cada vez que se invocara a un método, su código se convertiría en una pesadilla ilegible. Los programadores son reacios a admitir la verdad: este enfoque al manejo de errores es una grandísima limitación especialmente de cara a la creación de programas grandes, robustos y fácilmente mantenibles.

La solución es extraer del manejo de errores la naturaleza casual de los mismos y forzar la formalidad. Esto se ha venido haciendo a lo largo de bastante tiempo, dado que las implementaciones de *manejo de excepciones* se retornan hasta los sistemas operativos de los años 60, e incluso al "**error goto**" de BASIC. Pero el manejo de excepciones de C++ se basaba en Ada, y el de Java está basado fundamentalmente en C++ (aunque se parece incluso más al de Pascal Orientado a Objetos).

La palabra "excepción" se utiliza en el sentido de: "Yo me encargo de la excepción a eso". En cuanto se da un problema puede desconocerse qué hacer con el mismo, pero se sabe que simplemente no se puede continuar sin más; hay que parar y alguien, en algún lugar, deberá averiguar qué hacer. Pero puede que no se disponga de información suficiente en el contexto actual como para solucionar el problema. Por tanto, se pasa el problema a un contexto superior en el que alguien se pueda encargar de tomar la decisión adecuada (algo semejante a una cadena de comandos).

El otro gran beneficio de las excepciones es que limpian el código de manejo de errores. En vez de comprobar si se ha dado un error en concreto y tratar con él en diversas partes del programa, no es necesario comprobar nada más en el momento de invocar al método (puesto que la excepción ga-

---

[1] Como ejemplo de esta afirmación, un programador en C puede comprobar el valor que devolvía la función **printf( )**.

rantizará que alguien la capture). Y es necesario manejar el problema en un solo lugar, el denominado *gestor de excepciones*. Éste salva el código, y separa el código que describe qué se desea hacer a partir del código en ejecución cuando algo sale mal. En general, la lectura, escritura, y depuración de código se vuelve mucho más sencilla con excepciones, que cuando se hace uso de la antigua manera de gestionar los errores.

Debido a que el manejo de excepciones se ve fortalecido con el compilador de Java, hay numerosísimos ejemplos en este libro que permiten aprender todo lo relativo al manejo de excepciones. Este capítulo presenta el código que es necesario escribir para gestionar adecuadamente las excepciones, y la forma de generar excepciones si algún método se mete en problemas.

Excepciones básicas

Una *condición excepcional* es un problema que evita la continuación de un método o el alcance actual. Es importante distinguir una condición excepcional de un problema normal, en el que se tiene la suficiente información en el contexto actual como para hacer frente a la dificultad de alguna manera. Con una condición excepcional no se puede continuar el proceso porque no se tiene la información necesaria para tratar el problema, en el *contexto actual*. Todo lo que se puede hacer es salir del contexto actual y relegar el problema a un contexto superior. Esto es lo que ocurre cuando se lanza una excepción.

Un ejemplo sencillo es una división. Si se corre el riesgo de dividir entre cero, merece la pena comprobar y asegurarse de que no se seguirá adelante y se llegará a ejecutar la división. Pero ¿qué significa que el denominador sea cero? Quizás se sabe, en el contexto del problema que se está intentando solucionar en ese método particular, cómo manejar un denominador cero. Pero si se trata de un valor que no se esperaba, no se puede hacer frente a este error, por lo que habrá que lanzar una excepción en vez de continuar hacia delante.

Cuando se lanza una excepción, ocurren varias cosas. En primer lugar, se crea el objeto excepción de la misma forma en que se crea un objeto Java: en el montículo, con **new**. Después, se detiene el cauce normal de ejecución (el que no se podría continuar) y se lanza la referencia al objeto excepción desde el contexto actual. En este momento se interpone el mecanismo de gestión de excepciones que busca un lugar apropiado en el que continuar ejecutando el programa. Este lugar apropiado es el *gestor de excepciones*, cuyo trabajo es recuperarse del problema de forma que el programa pueda, o bien intentarlo de nuevo, o bien simplemente continuar.

Como un ejemplo sencillo de un lanzamiento de una excepción, considérese una referencia denominada **t**. Es posible que se haya recibido una referencia que no se haya inicializado, por lo que sería una buena idea comprobarlo antes de que se intentara invocar a un método utilizando esa referencia al objeto. Se puede enviar información sobre el error a un contexto mayor creando un objeto que represente la información y "arrojándolo" fuera del contexto actual. A esto se le llama *lanzamiento de una excepción*. Tiene esta apariencia:

```
if (t == null)
  throw new NullPointerException();
```

Esto lanza una excepción, que permite —en el contexto actual— abdicar la responsabilidad de pensar sobre este aspecto más adelante. Simplemente se gestiona automáticamente en algún otro sitio. El *dónde* se mostrará más tarde.

Parámetros de las excepciones

Como cualquier otro objeto en Java, las excepciones siempre se crean en el montículo haciendo uso de **new**, que asigna espacio de almacenamiento e invoca a un constructor. Hay dos constructores en todas las excepciones estándar: el primero es el constructor por defecto y el segundo toma un parámetro string de forma que se pueda ubicar la información pertinente en la excepción:

```
if (t == null)
  throw new NullPointerException("t = null");
```

Este string puede extraerse posteriormente utilizando varios métodos, como se verá más adelante.

La palabra clave **throw** hace que ocurran varias cosas relativamente mágicas. Habitualmente, se usará primero **new** para crear un objeto que represente la condición de error. Se da a **throw** la referencia resultante. En efecto, el método "devuelve" el objeto, incluso aunque ese tipo de objeto no sea el que el método debería devolver de forma natural. Una manera natural de pensar en las excepciones es como si se tratara de un mecanismo de retorno alternativo, aunque el que lleve esta analogía demasiado lejos acabará teniendo problemas. También se puede salir del ámbito ordinario lanzando una excepción. Pero se devuelve un valor, y el método o el ámbito finalizan.

Cualquier semejanza con un método de retorno ordinario acaba aquí, puesto que el punto de retorno es un lugar completamente diferente del punto al que se sale en una llamada normal a un método. (Se acaba en un gestor de excepciones adecuado que podría estar a cientos de kilómetros —es decir, mucho más bajo dentro de la pila de invocaciones— del punto en que se lanzó la excepción.)

Además, se puede lanzar cualquier tipo de objeto lanzable **Throwable** que se desee. Habitualmente, se lanzará una clase de excepción diferente para cada tipo de error. La información sobre cada error se representa tanto dentro del objeto excepción como implícitamente en el tipo de objeto excepción elegido, puesto que alguien de un contexto superior podría averiguar qué hacer con la excepción. (A menudo, la única información es el tipo de objeto excepción, y no se almacena nada significativo junto con el objeto excepción.)

Capturar una excepción

Si un método lanza una excepción, debe asumir que esa excepción será "capturada" y que será tratada. Una de las ventajas del manejo de excepciones de Java es que te permite concentrarte en el problema que se intenta solucionar en un único sitio, y tratar los errores que ese código genere en otro sitio.

Para ver cómo se captura una excepción, hay que entender primero el concepto de *región guardada*, que es una sección de código que podría producir excepciones, y que es seguida del código que maneja esas excepciones.

El bloque **try**

Si uno está dentro de un método y lanza una excepción (o lo hace otro método al que se invoque), ese método acabará en el momento en que haga el lanzamiento. Si no se desea que una **excepción** implique abandonar un método, se puede establecer un bloque especial dentro de ese método para que capture la excepción. A este bloque se le denomina el ***bloque try*** puesto que en él se "intentan" varias llamadas a métodos. El bloque try es un ámbito ordinario, precedido de la palabra clave **try**:

```
try {
  // Código que podría generar excepciones
}
```

Si se estuviera comprobando la existencia de errores minuciosamente en un lenguaje de programación que no soporte manejo de excepciones, habría que rodear cada llamada a método con código de prueba de invocación y errores, incluso cuando el mismo método fuese invocado varias veces. Esto significa que el código es mucho más fácil de escribir y leer debido a que no se confunde el objetivo del código con la comprobación de errores.

Manejadores de excepciones

Por supuesto, la excepción que se lance debe acabar en algún sitio. Este "sitio" es el *manejador de excepciones* y hay uno por cada tipo de excepción que se desee capturar. Los manejadores de excepciones siguen inmediatamente al bloque try y se identifican por la palabra clave **catch**:

```
try {
  // Código que podría generar excepciones
} catch(Tipo1 id1) {
  // Manejo de excepciones de Tipo1
} catch(Tipo2 id2) {
  // Manejo de excepciones de Tipo2
} catch(Tipo3 id3) {
  // Manejo de excepciones de Tipo3
}

// etc. . .
```

Cada cláusula *catch* (manejador de excepciones) es semejante a un pequeño método que toma uno y sólo un argumento de un tipo en particular. El identificador (**id1**, **id2**, y así sucesivamente) puede usarse dentro del manejador, exactamente igual que un parámetro de un método. En ocasiones nunca se usa el identificador porque el tipo de la excepción proporciona la suficiente información como para tratar la excepción, pero el identificador debe seguir ahí.

Los manejadores deben aparecer directamente tras el bloque *try*. Si se lanza una excepción, el mecanismo de gestión de excepciones trata de cazar el primer manejador con un argumento que coincida con el tipo de excepción. Posteriormente, entra en esa cláusula *catch*, y la excepción se da por manejada. La búsqueda de manejadores se detiene una vez que se ha finalizado la cláusula *catch*. Sólo se ejecuta la cláusula *catch*; no es como una sentencia **switch** en la que haya que colocar un **break** después de cada **case** para evitar que se ejecute el resto.

Fíjese que, dentro del bloque *try*, varias llamadas a métodos podrían generar la misma excepción, pero sólo se necesita un manejador.

Terminación o reanudación

Hay dos modelos básicos en la teoría de manejo de excepciones. En la *terminación* (que es lo que soportan Java y C++) se asume que el error es tan crítico que no hay forma de volver atrás a resolver dónde se dio la excepción. Quien quiera que lanzara la excepción decidió que no había forma de resolver la situación, y no *quería* volver atrás.

La alternativa es el *reanudación*. Significa que se espera que el manejador de excepciones haga algo para rectificar la situación, y después se vuelve a ejecutar el método que causó el error, presumiendo que a la segunda no fallará. Desear este segundo caso significa que se sigue pensando que la excepción continuará tras el manejo de la excepción. En este caso, la excepción es más como una llamada a un método —que es como deberían establecerse en Java aquellas situaciones en las que se desea este tipo de comportamiento. (Es decir, es mejor llamar a un método que solucione el problema antes de lanzar una excepción.) Alternativamente, se ubica el bloque **try** dentro de un bucle **while** que sigue intentando volver a entrar en el bloque **try** hasta que se obtenga el resultado satisfactorio.

Históricamente, los programadores que usaban sistemas operativos que soportaban el manejo de excepciones reentrantes acababan usando en su lugar código con terminación. Por tanto, aunque la técnica de los reintentos parezca atractiva a primera vista, no es tan útil en la práctica. La razón dominante es probablemente el *acoplamiento* resultante: el manejador debe ser, a menudo, consciente de dónde se lanza la excepción y contener el código no genérico específico del lugar de lanzamiento. Esto hace que el código sea difícil de escribir y mantener, especialmente en el caso de sistemas grandes en los que la excepción podría generarse en varios puntos.

Crear sus propias excepciones

No hay ninguna limitación que obligue a utilizar las excepciones existentes en Java. Esto es importante porque a menudo será necesario crear sus propias excepciones para indicar un error especial que puede crear su propia biblioteca, pero que no fue previsto cuando se creó la jerarquía de excepciones de Java.

Para crear su propia clase excepción, se verá obligado a heredar de un tipo de excepción existente, preferentemente uno cercano al significado de su nueva excepción (sin embargo, a menudo esto no es posible). La forma más trivial de crear un nuevo tipo de excepción es simplemente dejar que el compilador cree el constructor por defecto, de forma que prácticamente no haya que escribir ningún código:

```
//: c10:DemoExcepcionSencilla.java
// Heredando sus propias excepciones.
class ExcepcionSencilla extends Exception {}

public class DemoExcepcionSencillaDemo {
  public void f() throws ExcepcionSencilla {
    System.out.println(
      "Lanzando ExcepcionSencilla desde f()");
    throw new ExcepcionSencilla ();
  }
  public static void main(String[] args) {
    DemoExcepcionSencilla sed =
      new DemoExcepcionSencilla();
    try {
      sed.f();
    } catch(ExcepcionSencilla e) {
      System.err.println("¡Capturada!");
    }
  }
} ///:~
```

Cuando el compilador crea el constructor por defecto, se trata del que llama automáticamente (y de forma invisible) al constructor por defecto de la clase base. Por supuesto, en este caso no se obtendrá un constructor **ExcepcionSencilla(String)**, pero en la práctica esto no se usa mucho. Como se verá, lo más importante de una excepción es el nombre de la clase, por lo que en la mayoría de ocasiones una excepción como la mostrada arriba es plenamente satisfactoria.

Aquí, se imprime el resultado en la consola de *error estándar* escribiendo en **System.err**. Éste suele ser el mejor sitio para enviar información de error, en vez de **System.out**, que podría estar redirigida. Si se envía la salida a **System.err** no estará redireccionada junto con **System.out**, por lo que el usuario tiene más probabilidades de enterarse.

Crear una clase excepción que tenga también un constructor que tome un **String** como parámetro es bastante sencillo:

```
//: c10:ConstructoresCompletos.java
// Heredando tus propias excepciones.

class MiExcepcion extends Exception {
  public MiExcepcion() {}
  public MiExcepcion(String msg) {
    super(msg);
  }
}

public class ConstructoresCompletos {
  public static void f() throws MiExcepcion {
```

```
      System.out.println(
        "Lanzando MiExcepcion desde f()");
      throw new MiExcepcion();
    }
    public static void g() throws MiExcepcion {
      System.out.println(
        "Lanzando MiExcepcion desde g()");
      throw new MiExcepcion("Originada en g()");
    }
    public static void main(String[] args) {
      try {
        f();
      } catch(MiExcepcion e) {
        e.printStackTrace(System.err);
      }
      try {
        g();
      } catch(MiExcepcion e) {
        e.printStackTrace(System.err);
      }
    }
} ///:~
```

El código añadido es poco —la inserción de dos constructores que definen la forma de crear **MiExcepcion**. En el segundo constructor, se invoca explícitamente al constructor de la clase base con un parámetro **String** utilizando la palabra clave **super**.

Se envía a **System.err** información de seguimiento de la pila, de forma que habrá más probabilidades de que se haga notar en caso de que se haya redireccionado **System.out**.

La salida del programa es:

```
Lanzando MiExcepcion desde f()
MiExcepcion
    at FullConstructors.f(FullConstructors.java:16)
    at FullConstructors.main(FullConstructors.java:24)
Lanzando MiExcepcion desde g()
MiExcepcion: originada en g()
    at FullConstructors.g(FullConstructors.java:20)
    at FullConstructors.main(FullConstructors.java:29)
```

Se puede ver la ausencia del mensaje de detalle en la **MiExcepcion** lanzada desde **f( )**.

Se puede llevar aún más lejos el proceso de creación de nuevas excepciones. Se pueden añadir constructores y miembros extra:

```
//: c10:CaracteristicasExtra.java
```

```
// Embellecimiento aún mayor de las clases excepción.

class MiExcepcion2 extends Exception {
  public MiExcepcion2() {}
  public MiExcepcion2(String msg) {
    super(msg);
  }
  public MiExcepcion2(String msg, int x) {
    super(msg);
    i = x;
  }
  public int val() { return i; }
  private int i;
}

public class CaracteristicasExtra {
  public static void f() throws MiExcepcion2 {
    System.out.println(
      "Lanzando MiExcepcion2 desde f()");
    throw new MiExcepcion2();
  }
  public static void g() throws MiExcepcion2 {
    System.out.println(
      "Lanzando MiExcepcion2 desde g()");
    throw new MiExcepcion2("Originada en g()");
  }
  public static void h() throws MiExcepcion2 {
    System.out.println(
      "Lanzando MiExcepcion2 desde h()");
    throw new MiExcepcion2(
      "Originada en h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MiExcepcion2 e) {
      e.printStackTrace(System.err);
    }
    try {
      g();
    } catch(MiExcepcion2 e) {
      e.printStackTrace(System.err);
    }
    try {
      h();
```

```
    } catch(MiExcepcion2 e) {
      e.printStackTrace(System.err);
      System.err.println("e.val() = " + e.val());
    }
  }
} ///:~
```

Se ha añadido un dato miembro **i**, junto con un método que lee ese valor y un constructor adicional. La salida es:

```
Lanzando MiExcepcion2 desde f()
MiExcepcion2
  at CaracteristicasExtra.f(CaracteristicasExtra.java:22)
  at CaracteristicasExtra.main(CaracteristicasExtra.java:34)
Lanzando MyException2 desde g()
MyException2: Originada en g()
  at CaracteristicasExtra.g(CaracteristicasExtra.java:26)
  at CaracteristicasExtra.main(CaracteristicasExtra.java:39)
Lanzando MyException2 desde h()
MyException2: Originada en h()
  at CaracteristicasExtra.h(CaracteristicasExtra.java:30)
  at CaracteristicasExtra.main(CaracteristicasExtra.java:44)
e.val() = 47
```

Dado que una excepción es simplemente otro tipo de objeto, se puede continuar este proceso de embellecimiento del poder de las clases **excepción**. Hay que tener en cuenta, sin embargo, que todo este disfraz podría perderse en los programadores clientes que hagan uso de los paquetes, puesto que puede que éstos simplemente busquen el lanzamiento de la excepción, sin importarles nada más. (Ésta es la forma en que se usan la mayoría de las excepciones de biblioteca de Java.)

La especificación de excepciones

En Java, se pide que se informe al programador cliente, que llama al método, de las excepciones que podría lanzar ese método. Esto es bastante lógico porque el llamador podrá saber exactamente el código que debe escribir si desea capturar todas las excepciones potenciales. Por supuesto, si está disponible el código fuente, el programador cliente podría simplemente buscar sentencias **throw**, pero a menudo las bibliotecas no vienen con sus fuentes. Para evitar que esto sea un problema, Java proporciona una sintaxis (y *fuerza* el uso de la misma) para permitir decir educadamente al programador cliente qué excepciones lanza ese método, de forma que el programador cliente pueda manejarlas. Ésta es la *especificación de excepciones*, y es parte de la declaración del método, y se sitúa justo después de la lista de parámetros.

La especificación de excepciones utiliza la palabra clave **throws**, seguida de la lista de todos los tipos de excepción potenciales. Por tanto, la definición de un método podría tener la siguiente apariencia:

```
void f() throws DemasiadoGrande, DemasiadoPequeño, DivPorCero { // ...
```

Si se dice

```
void f() { // ...
```

significa que el método no lanza excepciones. (*Excepto* las excepciones de tipo **RuntimeException**, que puede ser lanzado razonablemente desde cualquier sitio —como se describirá más adelante.)

No se puede engañar sobre una especificación de excepciones —si un método provoca excepciones y no las maneja, el compilador lo detectará e indicará que, o bien hay que manejar la excepción o bien hay que indicar en la especificación de excepciones todas las excepciones que el método puede lanzar. Al fortalecer las especificaciones de excepciones de arriba abajo, Java garantiza que se puede asegurar la corrección de la excepción en *tiempo de compilación*[2].

Sólo hay un lugar en el que se pude engañar: se puede decir que se lanza una excepción que verdaderamente no se lanza. El compilador cree en tu palabra, y fuerza a los usuarios del método a tratarlo como si verdaderamente arrojara la excepción. Esto tiene un efecto beneficioso al ser un objeto preparado para esa excepción, de forma que, de hecho, se puede empezar a lanzar la excepción más tarde sin que esto requiera modificar el código ya existente. También es importante para la creación de clases base **abstractas** e **interfaces** cuyas clases derivadas o implementaciones pueden necesitar lanzar excepciones.

Capturar cualquier excepción

Es posible crear un manejador que capture cualquier tipo de excepción. Esto se hace capturando la excepción de clase base **Exception** (hay otros tipos de excepciones base, pero **Exception** es la clase base a utilizar pertinentemente en todas las actividades de programación):

```
catch(Exception e) {
  System.err.println("Excepcion capturada");
}
```

Esto capturará cualquier excepción, de forma que si se usa, habrá que ponerlo al *final* de la lista de manejadores para evitar que los manejadores de excepciones que puedan venir después queden ignorados.

Dado que la clase **Exception** es la base de todas las clases de excepción que son importantes para el programador, no se logra mucha información específica sobre la excepción, pero se puede llamar a los métodos que vienen de *su* tipo base **Throwable**:

String getMessage( )
String getLocalizedMessage( )
Toma el mensaje de detalle, o un mensaje ajustado a este escenario particular.

[2] Esto constituye una mejora significativa frente al manejo de excepciones de C++, que no captura posibles violaciones de especificaciones de excepciones hasta tiempo de ejecución, donde no es ya muy útil.

String toString( )
Devuelve una breve descripción del objeto *Throwable*, incluyendo el mensaje de detalle si es que lo hay.

void printStackTrace( )
void printStackTrace(PrintStream)
void printStackTrace(PrintWriter)
Imprime el objeto y la traza de pila de llamadas lanzada. La pila de llamadas muestra la secuencia de llamadas al método que condujeron al momento en que se lanzó la excepción. La primera versión imprime en el error estándar, la segunda y la tercera apuntan a un flujo de datos de tu elección (en el Capítulo 11, se entenderá por qué hay dos tipos de flujo de datos).

Throwable fillInStackTrace( )
Registra información dentro de este objeto **Throwable**, relativa al estado actual de las pilas. Es útil cuando una aplicación está relanzando un error o una excepción (en breve se contará algo más al respecto).

Además, se pueden conseguir otros métodos del tipo base de **Throwable**, **Object** (que es el tipo base de todos). El que podría venir al dedillo para excepciones es **getClass( )** que devuelve un objeto que representa la clase de este objeto. Se puede también preguntar al objeto de esta **Clase** por su nombre haciendo uso de **getName( )** o **toString( )**. Asimismo se pueden hacer cosas más sofisticadas con objetos **Class** que no son necesarios en el manejo de excepciones. Los objetos **Class** se estudiarán más adelante.

He aquí un ejemplo que muestra el uso de los métodos básicos de **Exception**:

```
//: c10:MetodosDeExcepcion.java
// Demostrando los métodos Exception.

public class MetodosDeExcepcion {
  public static void main(String[] args) {
    try {
      throw new Exception("Aquí esta mi excepcion");
    } catch(Exception e) {
      System.err.println("Excepcion capturada");
      System.err.println(
        "e.getMessage(): " + e.getMessage());
      System.err.println(
        "e.getLocalizedMessage(): " +
         e.getLocalizedMessage());
      System.err.println("e.toString(): " + e);
      System.err.println("e.printStackTrace():");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

La salida de este programa es:

```
Excepcion capturada
e.getMessage(): Aquí esta mi excepcion
e.getLocalizedMessage(): Aquí esta mi excepcion
e.toString(): java.lang.Exception:
  Aquí esta mi excepcion
e.printStackTrace():
java.lang.Exception: Aquí esta mi excepcion
 at ExceptionMethods.main(MetodosDeExcepcion.java:7)
java.lang.Exception:
  Aquí esta mi excepcion
 at MetodosDeExcepcion.main(Metodos de Excepcion.java:7)
```

Se puede ver que los métodos proporcionan más información exitosamente —cada una es efectivamente, un superconjunto de la anterior.

Relanzar una excepción

En ocasiones se desea volver a lanzar una excepción que se acaba de capturar, especialmente cuando se usa **Exception** para capturar cualquier excepción. Dado que ya se tiene la referencia a la excepción actual, se puede volver a lanzar esa referencia:

```
catch(Exception e) {
  System.err.println("Se ha lanzado una excepcion");
  throw e;
}
```

Volver a lanzar una excepción hace que la excepción vaya al contexto inmediatamente más alto de manejadores de excepciones. Cualquier cláusula **catch** subsiguiente del mismo bloque **try** seguirá siendo ignorada. Además, se preserva todo lo relativo al objeto, de forma que el contexto superior que captura el tipo de excepción específico pueda extraer toda la información de ese objeto.

Si simplemente se vuelve a lanzar la excepción actual, la información que se imprime sobre esa excepción en **printStackTrace( )** estará relacionada con el origen de la excepción, no al lugar en el que se volvió a lanzar. Si se desea instalar nueva información de seguimiento de la pila, se puede lograr mediante **fillInStackTrace( )**, que devuelve un objeto excepción creado rellenando la información de la pila actual en el antiguo objeto excepción. Éste es su aspecto:

```
//: c10:Relanzando.java
// Demostrando fillInStackTrace()

public class Relanzando {
  public static void f() throws Exception {
    System.out.println(
      "originando la excepcion en f()");
```

```
      throw new Exception("lanzada desde f()");
   }
   public static void g() throws Throwable {
      try {
         f();
      } catch(Exception e) {
         System.err.println(
            "Dentro de g(), e.printStackTrace()");
         e.printStackTrace(System.err);
         throw e; // 17
         // throw e.fillInStackTrace(); // 18
      }
   }
   public static void
   main(String[] args) throws Throwable {
      try {
         g();
      } catch(Exception e) {
         System.err.println(
            "Capturada en main, e.printStackTrace()");
         e.printStackTrace(System.err);
      }
   }
} ///:~
```

Los números de línea importantes están marcados como comentarios con la línea 17 sin comentarios (como se muestra), la salida sería:

```
originando la excepcion en f()
Dentro de g(), e.printStackTrace()
java.lang.Exception: lanzada desde f()
   at Relanzando.f(Relanzando.java:8)
   at Relanzando.g(Relanzando.java:12)
   at Relanzando.main(Relanzando.java:24)
Capturada en el main, e.printStackTrace()
java.lang.Exception: arrojada desde f()
   at Relanzando.g(Relanzando.java:18)
   at Relanzando.g(Relanzando.java:12)
   at Relanzando.main(Relanzando.java:24)
```

De forma que la traza de la pila de excepciones siempre recuerda su punto de origen verdadero, sin que importe cuántas veces se relanza.

Considerando que la línea 17 sea comentario, y no lo sea la línea 18, se usa **fillInStackTrace( )**, siendo el resultado:

```
originando la excepcion en f()
```

```
Dentro de g(), e.printStackTrace()
java.lang.Exception: lanzada desde f()
   at Relanzando.f(Relanzando.java:8)
   at Relanzando.g(Relanzando.java:12)
   at Relanzando.main(Relanzando.java:24)
Capturada en el main, e.printStackTrace()
java.lang.Exception: arrojada desde f()
   at Relanzando.g(Relanzando.java:18)
   at Relanzando.main(Relanzando.java:24)
```

Debido a **fillInStackTrace( )**, la línea 18 se convierte en el nuevo punto de origen de la excepción.

La clase **Throwable** debe aparecer en la especificación de excepciones de **g( )** y **main( )** porque **fillInStackTrace( )** produce una referencia a un objeto **Throwable**. Dado que **Throwable** es una clase base de **Exception**, es posible conseguir un objeto que es un **Throwable** pero *no* un **Exception**, de forma que el manejador para **Exception** en el método **main( )** podría perderla. Para asegurarse de que todo esté en orden, el compilador fuerza una especificación de excepciones que incluya **Throwable**. Por ejemplo, la excepción del siguiente programa *no* se captura en el método **main( )**:

```
//: c10:LanzarFuera.java
public class LanzarFuera {
   public static void
   main(String[] args) throws Throwable {
      try {
         throw new Throwable();
      } catch(Exception e) {
         System.err.println("Capturada en main()");
      }
   }
} ///:~
```

También es posible volver a lanzar una excepción diferente de la capturada. Si se hace esto, se consigue un efecto similar al usar **fillInStackTrace( )** —la información sobre el origen primero de la excepción se pierde, y lo que queda es la información relativa al siguiente **throw**:

```
//: c10:RelanzarNueva.java
// Relanzar un objeto distinto
// del capturado.

class ExcepcionUna extends Exception {
   public ExcepcionUna(String s) { super(s); }
}

class ExcepcionDos extends Exception {
   public ExcepcionDos (String s) { super(s); }
}
```

```
public class RelanzaNuevo {
  public static void f() throws ExcepcionUna {
    System.out.println(
      "originando la excepcion en f()");
    throw new ExcepcionUna("lanzada desde f()");
  }
  public static void main(String[] args)
  throws ExcepcionDos {
    try {
      f();
    } catch(ExcepcionUna e) {
      System.err.println(
        "Capturada en el método main, e.printStackTrace()");
      e.printStackTrace(System.err);
      throw new Excepcion2("desde la main()");
    }
  }
} ///:~
```

La salida es:

```
originando la excepcion en f()
Capturada en la main, e.printStackTrace()
OneException: lanzada desde f()
  at RelanzamientoNuevo.f(RelanzamientoNuevo.java:17)
  at RelanzamientoNuevo.main(RelanzamientoNuevo.java.22)
Excepcion in thread "main" ExcepcionDos: desde la main()
  at RelanzamientoNuevo.main(Rethrow.java:27)
```

La excepción final sólo sabe que proviene de **main( )**, y no de **f( )**.

No hay que preocuparse nunca de limpiar la excepción previa, o cualquier otra excepción en este sentido. Todas son objetos basados en el montículo creados con **new**, por lo que el recolector de basura los limpia automáticamente.

Excepciones estándar de Java

La clase **Throwable** de Java describe todo aquello que se pueda lanzar en forma de excepción. Hay dos tipos generales de objetos **Throwable** ("tipos de" = "heredados de"). **Error** representa los errores de sistema y de tiempo de compilación que uno no se preocupa de capturar (excepto en casos especiales). **Exception** es el tipo básico que puede lanzarse desde cualquier método de las clases de la biblioteca estándar de Java y desde los métodos que uno elabore, además de incidencias en tiempo de ejecución. Por tanto, el tipo base que más interesa al programador es **Exception**.

La mejor manera de repasar las excepciones es navegar por la documentación HTML de Java que se puede descargar de *java.sun.com*. Merece la pena hacer esto simplemente para tomar un contacto inicial con las diversas excepciones, aunque pronto se verá que no hay nada especial entre las distintas excepciones, exceptuando el nombre. Además, Java cada vez tiene más excepciones; básicamente no tiene sentido imprimirlas en un libro. Cualquier biblioteca nueva que se obtenga de un tercero probablemente tendrá también sus propias excepciones. Lo que es importante entender es el concepto y qué es lo que se debe hacer con las excepciones.

La idea básica es que el nombre de la excepción representa el problema que ha sucedido; de hecho se pretende que el nombre de las excepciones sea autoexplicativo. Las excepciones no están todas ellas definidas en **java.lang**; algunas están creadas para dar soporte a otras bibliotecas como **util**, **net** e **io**. Así, por ejemplo, todas las excepciones de E/S se heredan de **java.io.IOException**.

El caso especial de **RuntimeException**

El primer ejemplo de este capítulo fue:

```
if (t == null)
  throw new NullPointerException();
```

Puede ser bastante horroroso pensar que hay que comprobar que cualquier referencia que se pase a un método sea **null** o no **null** (de hecho, no se puede saber si el que llama pasa una referencia válida). Afortunadamente, no hay que hacerlo —esto es parte de las comprobaciones estándares de tiempo de ejecución que hace Java, de forma que si se hiciera una llamada a una referencia **null**, Java lanzaría automáticamente una **NullPointerException**. Por tanto, el fragmento de código de arriba es totalmente superfluo.

Hay un grupo completo de tipos de excepciones en esta categoría. Se trata de excepciones que Java siempre lanza automáticamente, y no hay que incluirlas en las especificaciones de excepciones. Además, están convenientemente agrupadas juntas bajo una única clase base denominada **RuntimeException**, que es un ejemplo perfecto de herencia: establece una familia de tipos que tienen algunas características y comportamientos en común. Tampoco se escribe nunca una especificación de excepciones diciendo que un método podría lanzar una **RuntimeException**, puesto que se asume. Dado que indican fallos, generalmente una **RuntimeException** nunca se captura —se maneja automáticamente. Si uno se viera forzado a comprobar las excepciones de tipo **RuntimeException**, su código se volvería farragoso. Incluso aunque generalmente las **RuntimeExceptions** no se capturan, en los paquetes que uno construya se podría decidir lanzar algunas **RuntimeExceptions**.

¿Qué ocurre si estas excepciones no se capturan? Dado que el compilador no fortalece las especificaciones de excepciones en este caso, es bastante verosímil que una **RuntimeException** pudiera filtrar todo el camino hacia el exterior hasta el método **main( )** sin ser capturada. Para ver qué ocurre en este caso, puede probarse el siguiente ejemplo:

```
//: c10:NuncaCapturado.java
// Ignorando RuntimeExceptions.
```

```
public class NuncaCapturado {
  static void f() {
    throw new RuntimeException("Desde f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
  }
} ///:~
```

Ya se puede ver que una **RuntimeException** (y cualquier cosa que se herede de la misma) es un caso especial, puesto que el compilador no exige una especificación de excepciones para ellas.

La salida es:

```
Exception in thread "main"
java.lang.RuntimeException: Desde f()
  at NuncaCapturado.f(NuncaCapturado.java:9)
  at NuncaCapturado.g(NuncaCapturado.java:12)
  at NuncaCapturado.main(NuncaCapturado.java:15)
```

Por tanto la respuesta es: si una **excepción en tiempo de ejecución** consigue todo el camino hasta el método **main( )** sin ser capturada, se invoca a **printStackTrace( )** para esa excepción, y el programa finaliza su ejecución.

Debe tenerse en cuenta que sólo se pueden ignorar en un código propio las **excepciones en tiempo de ejecución**, puesto que el compilador obliga a realizar el resto de gestiones. El razonamiento es que una **excepción en tiempo de ejecución** representa un error de programación:

1. Un error que no se puede capturar (la recepción de una referencia **null** proveniente de un programador cliente por parte de un método, por ejemplo).

2. Un error que uno, como programador, debería haber comprobado en su código (como un **ArrayIndexOutOfBoundsException** en la que se debería haber comprobado el tamaño del array).

Se puede ver fácilmente el gran beneficio aportado por estas excepciones, puesto que éstas ayudan en el proceso de depuración.

Es interesante saber que no se puede clasificar el manejo de excepciones de Java como si fuera una herramienta de propósito específico. Aunque efectivamente está diseñado para manejar estos errores de tiempo de compilación que se darán por motivos externos al propio código, simplemente es esencial para determinados tipos de fallos de programación que el compilador no pueda detectar.

Limpiando con finally

A menudo hay algunos fragmentos de código que se desea ejecutar independientemente de que se lancen o no excepciones dentro de un bloque **try**. Esto generalmente está relacionado con operaciones distintas de recuperación de memoria (puesto que de esto ya se encarga el recolector de basura). Para lograr este efecto, se usa una cláusula **finally**[3] al final de todos los manejadores de excepciones. El esquema completo de una sección de manejo de excepciones es por consiguiente:

```
try {
  // La region protegida: Actividades peligrosas que
  // podrían lanzar A, B o C
} catch (A a1) {
  // Manejador para la situación A
} catch (B b1) {
  // Manejador para la situación B
} catch (C c1) {
  // Manejador para la situación C
} finally {
  // Actividades que se dan siempre
}
```

Para demostrar que la cláusula **finally** siempre se ejecuta, puede probarse el siguiente programa:

```
//: c10:EjecucionFinally.java
// La clausula finally se ejecuta siempre.

class ExcepcionTres extends Exception {}

public class EjecucionFinally {
  static int contador = 0;
  public static void main(String[] args) {
    while(true) {
      try {
        // El post-incremento es cero la primera vez:
        if(contador++ == 0)
          throw new ExcepcionTres();
        System.out.println("Sin excepcion");
      } catch(ExcepcionTres e) {
        System.err.println("ExcepcionTres");
      } finally {
        System.err.println("Inicio de clausula finally");
        if(contador == 2) break; // salida del "while"
      }
```

[3] El manejo de excepciones de C++ no tiene la cláusula **finally** porque confía en los destructores para lograr este tipo de limpieza.

```
    }
  }
} ///:~
```

Este programa también da una orientación para manejar el hecho de que las excepciones de Java (al igual que ocurre en C++) no permiten volver a ejecutar a partir del punto en que se lanzó la excepción, como ya se comentó. Si se ubica el bloque **try** dentro de un bloque, se podría establecer una condición a alcanzar antes de continuar con el programa. También se puede añadir un contador **estático** o algún otro dispositivo para permitir al bucle intentar distintos enfoques antes de rendirse. De esta forma se pueden construir programas de extremada fortaleza.

La salida es:

```
ExcepcionTres
Inicio de clausula finally
Sin excepcion
Inicio de clausula finally
```

Se lance o no una excepción, la cláusula **finally** se ejecuta siempre.

¿Para qué sirve **finally**?

En un lenguaje en el que no haya recolector de basura y sin llamadas automáticas a destructores[4], **finally** es importante porque permite al programador garantizar la liberación de memoria independientemente de lo que ocurra en el bloque **try**. Pero Java tiene recolección de basura, por lo que la liberación de memoria no suele ser un problema. Además, no tiene destructores a los que invocar. Así que, ¿cuándo es necesario usar **finally** en Java?

La cláusula **finally** es necesaria cuando hay que hacer algo *más* que devolver la memoria a su estado original. Éste es un tipo de limpieza equivalente a la apertura de un fichero o de una conexión de red, algo que seguro se habrá manejado más de una vez, y que se modela en el ejemplo siguiente:

```
//: c10:EncenderApagarInterrumpir.java
// ¿Por qué usar finally?

class Interrumpir {
  boolean estado = false;
  boolean leer() { return estado; }
  void encender() { estado = true; }
  void apagar() { estado = false; }
}
class ExcepcionEncenderApagar1 extends Exception {}
class ExcepcionEncenderApagar2 extends Exception {}
```

[4] Un destructor es una función a la que se llama siempre que se deja de usar un objeto. Siempre se sabe exactamente cuándo y dónde llamar al destructor. C++ tiene llamadas automáticas a destructores, pero las versiones 1 y 2 del Objeto Pascal de Delphi no (lo que cambia el significado y el uso del concepto de destructor en estos lenguajes).

```
public class EncenderApagarInterruptor {
   static Interruptor sw = new Interruptor();
   static void f() throws
      ExcepcionEncenderApagar1, ExcepcionEncenderApagar2 {}
   public static void main(String[] args) {
      try {
         sw.encender();
         // Código que puede lanzar excepciones...
         f();
         sw.apagar();
      } catch(OnOffException1 e) {
         System.err.println("OnOffException1");
         sw.apagar();
      } catch(ExcepcionEncenderApagar2 e) {
         System.err.println("ExcepcionEncenderApagar2");
         sw.apagar();
      }
   }
} ///:~
```

Aquí el objetivo es asegurarse de que el interruptor esté apagado cuando se complete el método **main( )**, por lo que **sw.apagar( )** se coloca al final del bloque try y al final de cada manejador de excepción. Pero es posible que se pudiera lanzar una excepción no capturada aquí, por lo que se perdería el **sw.apagar( )**. Sin embargo, con **finally**, se puede colocar código de limpieza simplemente en un lugar:

```
//: c10:ConFinally.java
// Finally garantiza la limpieza.

public class ConFinally {
   static Interruptor sw = new Interruptor();
   public static void main(String[] args) {
      try {
         sw.encender();
         // Código que puede lanzar excepciones...
         EncenderApagarInterruptor.f();
      } catch(ExcepcionEncenderApagar1 e) {
         System.err.println("OnOffException1");
      } catch(ExcepcionEncenderApagar2 e) {
         System.err.println("ExcepcionEncenderApagar2");
      } finally {
         sw.apagar();
      }
   }
} ///:~
```

Aquí se ha movido el **sw.apagar( )** simplemente un lugar, en el que se garantiza su ejecución independientemente de lo que pase.

Incluso en las clases en las que la excepción no se capture en el conjunto de cláusulas **catch**, se ejecutará **finally** antes de que el mecanismo de manejo de excepciones continúe su búsqueda de un manejador de nivel superior:

```
//: c10:SiempreFinally.java
// Finally se ejecuta siempre.

class ExceptionCuatro extends Exception {}

public class SiempreFinally {
  public static void main(String[] args) {
    System.out.println(
      "Entrando en el primer bloque try");
    try {
      System.out.println(
        "Entrando en el segundo bloque try");
      try {
        throw ExcepcionCuatro();
      } finally {
        System.out.println(
          "finally en el segundo bloque try");
      }
    } catch(ExcepcionCuatro e) {
      System.err.println(
        "Capturada ExcepcionCuatro en el primer bloque try");
    } finally {
      System.err.println(
        "finally en el primer bloque try");
    }
  }
} ///:~
```

La salida de este programa muestra lo que ocurre:

```
Entrando en el primer bloque try
Entrando en el segundo bloque try
finally en el segundo bloque try
Capturada ExcepcionCuatro en el primer bloque try
finally en el primer bloque try
```

La sentencia **finally** también se ejecutará en aquellos casos en que se vean involucradas sentencias **break** y **continue**. Fíjese que, con las sentencias **break** y **continue**, **finally** elimina la necesidad de una sentencia **goto** en Java.

Peligro: la excepción perdida

En general, la implementación de las excepciones en Java destaca bastante, pero desgraciadamente tiene un problema. Aunque las excepciones son una indicación de una crisis en un programa, y nunca debería ignorarse, es posible que simplemente se pierda una excepción. Esto ocurre con una configuración particular al hacer uso de la cláusula **finally**:

```
//: c10:MensajePerdido.java
// Cómo puede perderse una excepcion.

class ExceptionMuyImportante extends Exception {
  public String toString() {
    return "¡Una excepcion muy importante!";
  }
}

class ExcepcionTrival extends Exception {
  public String toString() {
    return "Una excepcion trivial";
  }
}

public class MensajePerdido {
  void f() throws ExcepcionMuyImportante {
    throw new ExcepcionMuyImportante();
  }
  void disponer() throws ExcepcionTrivial {
    throw new ExcepcionTrivial();
  }
  public static void main(String[] args)
      throws Exception {
    MensajePerdido lm = new MensajePerdido();
    try {
      lm.f();
    } finally {
      lm.disponer();
    }
  }
} ///:~
```

La salida es:

```
Exception in thread "main" Una excepcion trivial
  at MensajePerdido.disponer(MensajePerdido.java:21)
  at MensajePerdido.main(MensajePerdido.java:29)
```

Se puede ver que no hay evidencia de la **ExcepcionMuyImportante**, que simplemente es reemplazada por la **ExcepcionTrivial** en la cláusula **finally**. Ésta es una trampa bastante seria, puesto que significa que una excepción podría perderse completamente, incluso de forma más oculta y difícil de detectar que en el ejemplo de arriba. Por el contrario, C++ trata la situación en la que se lanza una segunda excepción antes de que se maneje la primera como un error de programación fatal. Quizás alguna versión futura de Java repare este problema (por otro lado, generalmente se envuelve todo método que lance alguna excepción, tal como **dispose( )** dentro de una cláusula **try-catch)**.

Restricciones a las excepciones

Cuando se superpone un método, sólo se pueden lanzar las excepciones que se hayan especificado en la versión de la clase base del método. Ésta es una restricción útil, pues significa que todo código que funcione con la clase base funcionará automáticamente con cualquier objeto derivado de la clase base (un concepto fundamental en POO, por supuesto), incluyendo las excepciones.

Este ejemplo demuestra los tipos de restricciones impuestas (en tiempo de compilación) a las excepciones:

```
//: c10:CarreraTormentosa.java
// Los métodos superpuestos sólo pueden lanzar las
// excepciones especificadas en su versión clase base,
// o excepciones derivadas de las excepciones de la
// clase base.

class ExcepcionBeisbol extends Exception {}
class Falta extends ExcepcionBeisbol {}
class Strike extends ExcepcionBeisbol {}

abstract class Carrera {
  Carrera() throws ExcepcionBeisbol {}
  void evento () throws ExcepcionBeisbol {
   // De hecho no tiene que lanzar nada
  }
  abstract void batear() throws Strike, Foul;
  void caminar() {} // No lanza nada
}

class ExcepcionTormenta extends Exception {}
class Llueve extends ExceptionTormenta {}
class Eliminacion extends Falta {}

interface Tormenta {
  void evento() throws Llueve;
  void lloverFuerte() throws Llueve;
}
```

```
public class CarreraTormentosa extends Carrera
    implements Tormenta {
  // Ok para añadir nuevas excepciones a
  // los constructores, pero hay que hacerlo
  // con las excepciones del constructor base:
  CarreraTormentosa() throws Llueve,
    ExcepcionBeisbol {}
  CarreraTormentosa(String s) throws Falta,
   Excepcion Beisbol {}
  // Los métodos normales deben estar conformes con
  // la clase base:
//! void caminar() throws eliminación {} //Error de compilación
  // Una Interfaz NO PUEDE añadir excepciones a los
  // métodos existentes de la clase base:
//! public void evento() throws Llueve {}
  // Si el método no existe aún en la clase base,
  // OK con la excepción:
  public void lloverFuerte() throws Llueve {}
  // Se puede elegir no lanzar excepciones,
  // incluso si lo hace la versión base:
  public void evento() {}
  // Los métodos superpuestos pueden lanzar
  // excepciones heredadas:
  void batear() throws Eliminacion {}
  public static void main(String[] args) {
    try {
      CarreraTormentosa si = new CarreraTormentosa();
      si.batear();
    } catch(Eliminacion e) {
      System.err.println("Eliminacion");
    } catch(Llueve e) {
      System.err.println("Llueve");
    } catch(ExcepcionBeisbol e) {
      System.err.println("Error generico");
    }
    // Strike no se lanza en la versión derivada.
    try {
      // ¿Qué ocurre si se hace una conversión hacia arriba?
      Carrera i = new CarreraTormentosa();
      i.batear();
      // Hay que capturar las excepciones desde
      // la versión clase base del método:
    } catch(Strike e) {
      System.err.println("Strike");
    } catch(Falta e) {
```

```
        System.err.println("Falta");
      } catch(Llueve e) {
        System.err.println("Llueve");
      } catch(ExcepcionBeisbol e) {
        System.err.println(
          "Excepcion beisbol generica");
      }
    }
} ///:~
```

En **Carrera**, se puede ver que tanto el método **evento( )** como el constructor dicen que lanzarán una excepción, pero nunca lo hacen. Esto es legal porque permite forzar al usuario a capturar cualquier excepción que se pueda añadir a versiones superpuestas de **evento( )**. Como se ve en **batear( )**, en el caso de métodos **abstractos** se mantiene la misma idea.

La **interfaz Tormenta** es interesante porque contiene un método (**evento( )**) que está definido en **Carrera**, y un método que no lo está. Ambos métodos lanzan un nuevo tipo de excepción, **llueve**. Cuando **CarreraTormentosa hereda de carrera** e **implementa Tormenta**, se verá que el método **evento( )** de **Tormenta** *no puede* cambiar la interfaz de excepciones de **evento( )** en **Carrera**. De nuevo, esto tiene sentido porque de otra forma nunca se sabría si se está capturando lo correcto al funcionar con la clase base. Por supuesto, si un método descrito en una **interfaz** no está en la clase base, como ocurre con **lloverFuerte( )**, entonces no hay problema si lanza excepciones.

La restricción sobre las excepciones no se aplica a los constructores. En **CarreraTormentosa** se puede ver que un constructor puede lanzar lo que desee, independientemente de lo que lance el constructor de la clase base. Sin embargo, dado que siempre se llamará de una manera u otra a un constructor de clase base (aquí se llama automáticamente al constructor por defecto), el constructor de la clase derivada debe declarar cualquier excepción del constructor de la clase base en su especificación de excepciones. Fíjese que un constructor de clase derivada no puede capturar excepciones lanzadas por el constructor de su clase base.

La razón por la que **CarreraTormentosa.caminar( )** no compilará es que lanza una excepción, mientras que **Carrera.caminar( )** no lo hace. Si se permitiera esto, se podría escribir código que llamara a **Carrera.caminar( )** y que no tuviera que manejar ninguna excepción, pero entonces, al sustituir un objeto de una clase derivada de **Carrera** se lanzarían excepciones, causando una ruptura del código. Forzando a los métodos de la clase derivada a ajustarse a las especificaciones de excepciones de los métodos de la clase base, se mantiene la posibilidad de sustituir objetos.

El método **evento( )** superpuesto muestra que una versión de clase derivada de un método puede elegir no lanzar excepciones, incluso aunque lo haga la versión de clase base. De nuevo, esto es genial, puesto que no rompe ningún código escrito —asumiendo que la versión de clase base lanza excepciones. A **batear( )** se le aplica una lógica semejante, pues ésta lanza **Eliminación**, una excepción derivada de **Falta**, lanzada por la versión de clase base de **batear( )**. De esta forma, si alguien escribe código que funciona con **Carrera** y llama a **batear( )**, debe capturar la excepción **Falta**. Dado que **Eliminación** deriva de **Falta**, el manejador de excepciones también capturará **Eliminación**.

El último punto interesante está en el método **main( )**. Aquí se puede ver que si se está tratando con un objeto **CarreraTormentosa**, el compilador te fuerza a capturar sólo las excepciones específicas a esa clase, pero si se hace un conversión hacia arriba al tipo base, el compilador te fuerza (correctamente) a capturar las excepciones del tipo base. Todas estas limitaciones producen un código de manejo de excepciones más robusto[5].

Es útil darse cuenta de que aunque las especificaciones de excepciones se ven reforzadas por el compilador durante la herencia, las especificaciones de excepciones no son parte del tipo de un método, que está formado sólo por el nombre del método y los tipos de parámetros. Además, justo porque existe una especificación de excepciones en una versión de clase base de un método, no tiene por qué existir en la versión de clase derivada del mismo. Esto es bastante distinto de lo que dictaminan las reglas de herencia, según las cuales todo método de la clase base debe existir también en la clase derivada. Dicho de otra forma, "la interfaz de especificación de excepciones" de un método particular puede estrecharse durante la herencia y superponerse, pero no puede ancharse —esto es precisamente lo contrario de la regla de la interfaz de clases durante la herencia.

Constructores

Cuando se escribe código con excepciones, es particularmente importante que siempre se pregunte: "Si se da una excepción, ¿será limpiada adecuadamente?" La mayoría de veces es bastante seguro, pero en los constructores hay un problema. El constructor pone el objeto en un estado de partida seguro, pero podría llevar a cabo otra operación —como abrir un fichero— que no se limpia hasta que el usuario haya acabado con el objeto y llame a un método de limpieza especial. Si se lanza una excepción desde dentro de un constructor, puede que estos comportamientos relativos a la limpieza no se den correctamente. Esto significa que hay que ser especialmente cuidadoso al escribir constructores.

Dado que se acaba de aprender lo que ocurre con **finally**, se podría pensar que es la solución correcta. Pero no es tan simple, puesto que **finally** ejecuta *siempre* el código de limpieza, incluso en las situaciones en las que no se desea que se ejecute este código de limpieza hasta que acabe el método de limpieza. Por consiguiente, si se lleva a cabo una limpieza en **finally**, hay que establecer algún tipo de indicador cuando el constructor finaliza normalmente, de forma que si el indicador está activado no se ejecute nada en **finally**. Dado que esto no es especialmente elegante (se está asociando el código de un sitio a otro), es mejor si se intenta evitar llevar a cabo este tipo de limpieza en el método **finally**, a menos que uno se vea forzado a ello.

En el ejemplo siguiente, se crea una clase llamada **ArchivoEntrada** que abre un archivo y permite leer una línea (convertida a **String**) de una vez. Usa las clases **FileReader** y **BufferedReader** de la biblioteca estándar de E/S de Java que se verá en el Capítulo 11, pero que son lo suficientemente simples como para no tener ningún problema en tener su uso básico:

---

[5] ISO C++ añadió limitaciones semejantes que requieren que las excepciones de métodos derivados sean las mismas, o derivadas, de las excepciones que lanza el método de la clase base. Éste es un caso en el que C++, de hecho, puede comprobar las especificaciones de excepciones en tiempo de compilación.

```
//: c10:Limpieza.java
// Prestando atención a las excepciones
// en los constructores.
import java.io.*;

class ArchivoEntrada {
  private BufferedReader entrada;
  ArchivoEntrada(String nombref) throws Exception {
    try {
      in =
        new BufferedReader(
          new FileReader(nombref));
      // Otro código que podría lanzar excepciones
    } catch(FileNotFoundException e) {
      System.err.println(
        "No se pudo abrir " + nombref);
      // No se abrió, así que no se cierra
      throw e;
    } catch(Exception e) {
      // Todas las demás excepciones deben cerrarlo
      try {
        entrada.close();
      } catch(IOException e2) {
        System.err.println(
          "entrada.close() sin exito");
      }
      throw e; // Relanzar
    } finally {
      // ¡¡¡No cerrarlo aquí!!!
    }
  }
  String obtenerLinea() {
    String s;
    try {
      s = entrada.readLine();
    } catch(IOException e) {
      System.err.println(
        "obtenerLinea() sin exito");
      s = "fallo";
    }
    return s;
  }
  void limpiar() {
    try {
      entrada.close();
```

```
      } catch(IOException e2) {
        System.err.println(
          "entrada.close() sin exito");
      }
    }
  }

public class Limpieza {
  public static void main(String[] args) {
    try {
      ArchivoEntrada entrada =
        new ArchivoEntrada("Limpieza.java");
      String s;
      int i = 1;
      while((s = entrada.obtenerLinea()) != null)
        System.out.println(""+ i++ + ": " + s);
      entrada.limpiar();
    } catch(Exception e) {
      System.err.println(
        "Capturando en el método main, e.printStackTrace()");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

El constructor de **ArchivoEntrada** toma un parámetro **String**, que es el nombre del archivo que se desea abrir. Dentro de un bloque **try**, crea un **FileReader** usando el nombre de archivo. Un **FileReader** no es particularmente útil hasta que se usa para crear un **BufferedReader** con el que nos podemos comunicar —nótese que uno de los beneficios de **ArchivoEntrada** es que combina estas dos acciones.

Si el constructor **FileReader** no tiene éxito, lanza una **FileNotFoundException** que podría ser capturada de forma separada porque éste es el caso en el que no se quiere cerrar el archivo, puesto que éste no se abrió con éxito. Cualquier *otra* cláusula de captura debe cerrar el archivo, puesto que *fue* abierto en el momento en que se entra en la cláusula **catch**. (Por supuesto, esto es un truco si **FileNotFoundException** puede ser lanzado por más de un método. En este caso, se podría desear romper todo en varios bloques **try**.) El método **close( )** podría lanzar una excepción, por lo que es probado y capturado incluso aunque se encuentra dentro de otro bloque de otra cláusula **catch** —es simplemente otro par de llaves para el compilador de Java. Después de llevar a cabo operaciones locales, se relanza la excepción, lo cual es apropiado porque este constructor falló, y no se desea llamar al método asumiendo que se ha creado el objeto de manera adecuada o que sea válido.

En este ejemplo, que no usa la técnica de los indicadores anteriormente mencionados, la cláusula **finally** *no* es definitivamente el lugar en el que **close( )** (cerrar) el archivo, puesto que lo cerraría cada vez que se complete el constructor. Dado que queremos que se abra el fichero durante la vida útil del objeto **ArchivoEntrada** esto no sería apropiado.

EL método **obtenerLinea( )** devuelve un **String** que contiene la línea siguiente del archivo. Llama a **leerLinea( )**, que puede lanzar una excepción, pero esa excepción se captura de forma que **obtenerLinea( )** no lanza excepciones. Uno de los aspectos de diseño de las excepciones es la decisión de si hay que manejar una excepción completamente en este nivel, o hay que manejarla parcialmente y pasar la misma excepción (u otra), o bien simplemente pasarla. Pasarla, siempre que sea apropiada, puede definitivamente simplificar la codificación. El método **obtenerLinea( )** se convierte en:

```
String obtenerLinea() throws IOException {
  return entrada.readLine();
}
```

Pero por supuesto, el objeto que realiza la llamada es ahora el responsable de manejar cualquier **IOException** que pudiera surgir.

El usuario debe llamar al método **limpiar( )** al acabar de usar el objeto **ArchivoEntrada**. Esto liberará los recursos del sistema (como los manejadores de archivos) que fueron usados por el **BufferedReader** y/u objetos **FileReader**[6]. Esto no se quiere hacer hasta que se acabe con el objeto **InputFile**, en el momento en el que se le deje marchar. Se podría pensar en poner esta funcionalidad en un método **finalize( )**, pero como se mencionó en el Capítulo 4, no se puede estar seguro de que se invoque siempre a **finalize( )** (incluso si se *puede* estar seguro de que se invoque, no se sabe *cuándo*). Éste es uno de los puntos débiles de Java: toda la limpieza —que no sea la limpieza de memoria— no se da automáticamente, por lo que hay que informar al programador cliente de que es responsable, y debe garantizar que se dé la limpieza usando **finalize( )**.

En **Limpieza.java** se crea un **ArchivoEntrada** para abrir el mismo archivo fuente que crea el programa, se lee el archivo de línea en línea, y se añaden números de línea. Se capturan de forma genérica todas las excepciones en el método **main( )**, aunque se podría elegir una granularidad mayor.

Uno de los beneficios de este ejemplo es mostrar por qué se presentan las excepciones en este punto del libro —no se puede hacer E/S básica sin usar las excepciones. Las excepciones son tan integrantes de la programación de Java, especialmente porque el compilador las fortalece, que se puede tener éxito si no se conoce bien cómo trabajar con ellas.

Emparejamiento de excepciones

Cuando se lanza una excepción, el sistema de manejo de excepciones busca en los manejadores más "cercanos" en el mismo orden en que se escribieron. Cuando hace un emparejamiento, se considera que la excepción ya está manejada y no se llevan a cabo más búsquedas.

Emparejar una excepción no exige un proceso perfecto entre la excepción y su manejador. Un objeto de clase derivada se puede emparejar con un manejador de su clase base, como se ve en el ejemplo siguiente:

---

[6] En C++ un destructor se encargaría de esto por ti.

```
//: c10:Humano.java
// Capturando jerarquías de excepciones.

class Molestia extends Exception {}
class Estornudo extends Molestia {}

public class Humano {
  public static void main(String[] args) {
    try {
      throw new Estornudo();
    } catch(Estornudo s) {
      System.err.println("Estornudo capturado");
    } catch(Molestia a) {
      System.err.println("Molestia capturada");
    }
  }
} ///:~
```

La excepción **Estornudo** será capturada por la primera cláusula **catch** con la que se empareje —que es la primera, por supuesto. Sin embargo, si se anula la primera cláusula **catch** dejando sólo:

```
    try {
      throw new Estornudo();
    } catch(Molestia a) {
      System.err.println("Molestia capturada");
    }
```

El código seguirá funcionando porque está capturando la clase base de **Estornudo**. Dicho de otra forma, **catch(Molestia e)** capturará una **Molestia** o *cualquier otra clase derivada de ésta*. Esto es útil porque si se decide añadir nuevas excepciones derivadas a un método, el código del programador cliente no necesitará ninguna modificación mientras el cliente capture las excepciones de clase base.

Si se intenta "enmascarar" las excepciones de la clase derivada poniendo la cláusula **catch** de la clase base la primera, como en:

```
    try {
      throw new Estornudo();
    } catch(Molestia a) {
      System.err.println("Molestia capturada");
    } catch(Estornudo s) {
      System.err.println("Estornudo capturado");
    }
```

el compilador dará un mensaje de error, puesto que ve que nunca se alcanzará la cláusula catch de **Estornudo**.

Guías de cara a las excepciones

Utilice excepciones para:

1. Arreglar el problema y llamar de nuevo al método que causó la excepción.
2. Arreglar todo y continuar sin volver a ejecutar el método.
3. Calcular algún resultado alternativo en vez de lo que se suponía que iba a devolver el método.
4. Hacer lo que se pueda en el contexto actual y relanzar la *misma* excepción a un contexto superior.
5. Hacer lo que se pueda en el contexto actual y lanzar una excepción *diferente* a un contexto superior.
6. Terminar el programa.
7. Simplificar. (Si tu esquema de excepción hace algo más complicado, es una molestia utilizarlo.)
8. Hacer más seguros la biblioteca y el programa. (Ésta es una inversión a corto plazo de cara a la depuración, y también una inversión a largo plazo de cara a la fortaleza de la aplicación.)

Resumen

La recuperación mejorada de errores es una de las formas más poderosas de incrementar la fortaleza del código. La recuperación de errores es fundamental en todos los programas que se escriban, pero es especialmente importante en Java, donde uno de los objetivos principales es crear componentes de programa para que otros los usen. *Para crear un sistema robusto, cada componente debe ser robusto.*

Los objetivos del manejo de excepciones en Java son simplificar la creación de programas grandes y seguros utilizando menos código de lo actualmente disponible, y con garantías de que la aplicación no tenga errores sin manejar.

Las excepciones no son terribles de aprender, y son una de esas características que proporcionan beneficios inmediatos y significativos al proyecto. Afortunadamente, Java fortalece todos los aspectos de las excepciones, por lo que se garantiza que se usarán consistentemente por parte tanto de los diseñadores de bibliotecas como de programadores cliente.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *www.BruceEckel.con*.

1. Crear una clase con un método **main( )** que lance un objeto de tipo **Exception** dentro de un bloque **try**. Dar al constructor de **Exception** un parámetro **String**. Capturar la excepción en una cláusula **catch** e imprimir el parámetro **String**. Añadir una cláusula **finally** e imprimir un mensaje para probar que uno paso por ahí.

2. Crear una clase de excepción usando la palabra clave **extends**. Escribir un constructor para esta clase que tome un parámetro **String** y lo almacene dentro del objeto con una referencia **String**. Escribir un método que imprima el **String** almacenado. Crear una cláusula **try-catch** para probar la nueva excepción.

3. Escribir una clase con un método que lance una excepción del tipo creado en el Ejercicio 2. Intentar compilarla sin especificación de excepción para ver qué dice el compilador. Añadir la especificación apropiada. Probar la clase y su excepción dentro de una cláusula try-catch.

4. Definir una referencia a un objeto e inicializarla a **null**. Intentar llamar a un método mediante esta referencia. Ahora envolver el código en una cláusula **try-catch** para capturar la excepción.

5. Crear una clase con dos métodos, **f( )** y **g( )**. En **g( )**, lanzar una excepción de un nuevo tipo a definir. En **f( )**, invocar a **g( )**, capturar su excepción y, en la cláusula **catch**, lanzar una excepción distinta (de tipo diferente al definido). Probar el código en un método **main( )**.

6. Crear tres nuevos tipos de excepciones. Escribir una clase con un método que lance las tres. En el método **main( )**, invocar al método pero usar sólo una única cláusula **catch** para capturar los tres tipos de excepción.

7. Escribir código para generar y capturar una **ArrayIndexOutOfBoundsException**.

8. Crear tu propio comportamiento reiniciador utilizando un bucle **while** que se repita hasta que se deje de lanzar una excepción.

9. Crear una jerarquía de excepciones de tres niveles. Crear a continuación una clase base **A** con un método que lance una excepción a la base de la jerarquía. Heredar **B** de **A** y superponer el método de forma que lance una excepción en el nivel dos de la jerarquía. Repetir heredando la clase **C** de **B**. En el método **main( )**, crear un objeto de tipo **C** y hacer un conversión hacia arriba a **A**. Después, llamar al método.

10. Demostrar que un constructor de clase derivada no puede capturar excepciones lanzadas por el constructor de su clase base.

11. Mostrar que **EncenderApagarInterruptor.java** puede fallar lanzando una **RuntimeException** dentro del bloque **try**.

12. Mostrar que **ConFinally.java** no falla lanzando una **RuntimeException** dentro de un bloque **try**.

13. Modificar el Ejercicio 6 añadiendo una cláusula **finally**. Mostrar que esta cláusula **finally** se ejecuta, incluso si se lanza una **NullPointerException**.

14. Crear un ejemplo en el que se use un indicador para controlar si se llama a código de limpieza, tal y como se comentó en el segundo párrafo del epígrafe "Constructores".

15. Modificar **CarreraTormentosa.java** añadiendo un tipo de excepción **ArgumentoArbitro** y métodos que la lancen. Probar la jerarquía modificada.

16. Eliminar la primera cláusula **catch** de **Humano.java** y verificar que el código se sigue ejecutando y compilando correctamente.

17. Añadir un segundo nivel de pérdida de excepciones a **MensajePerdido.java** de forma que la propia **ExcepcionTrivial** se vea reemplazada por una tercera excepción.

18. En el Capítulo 5, encontrar los dos programas denominados **Afirmacion.java** y modificarlos de forma que lancen su propio tipo de excepción en vez de imprimir a **System.err**. Esta excepción debería estar en una clase interna que extienda **RuntimeException**.

19. Añadir un conjunto apropiado de excepciones a **c08:ControlesInvernadero.java**.

11: El sistema de E/S de Java

Crear un buen sistema de entrada/salida (E/S) es una de las tareas más difíciles para el diseñador de un lenguaje.

Esto es evidente con sólo observar la cantidad de enfoques diferentes. El reto parece estar en cubrir todas las posibles eventualidades. No sólo hay distintas fuentes y consumidores de información de E/S con las que uno desea comunicarse (archivos, la consola, conexiones de red), pero hay que comunicarse con ellos de varias maneras (secuencial, acceso aleatorio, espacio de almacenamiento intermedio, binario, carácter, mediante líneas, mediante palabras, etc.).

Los diseñadores de la biblioteca de Java acometieron este problema creando muchas clases. De hecho, hay tantas clases para el sistema de E/S de Java que puede intimidar en un principio (irónicamente, el diseño de la E/S de Java evita una explosión de clases). También hubo un cambio importante en la biblioteca de Java después de la versión 1.0, al suprimir la biblioteca original orientada a **bytes** por clases de E/S orientadas a **char** basadas en Unicode. Como resultado hay que aprender un número de clases aceptable antes de entender suficientemente un esbozo de la E/S de Java para poder usarla adecuadamente. Además, es bastante importante entender la historia de la biblioteca de E/S, incluso si tu primera reacción es: "¡No me aburras con esta historia, simplemente dime cómo usarla!" El problema es que sin la perspectiva histórica es fácil confundirse con algunas de las clases, y no comprender cuándo deberían o no usarse.

Este capítulo presentará una introducción a la variedad de clases de E/S contenidas en la biblioteca estándar de Java, y cómo usarlas.

La clase **File**

Antes de comenzar a ver las clases que realmente leen y escriben datos en flujos, se echará un vistazo a una utilidad proporcionada por la biblioteca para manejar aspectos relacionados con directorios de archivos.

La clase **File** tiene un nombre engañoso —podría pensarse que hace referencia a un archivo, pero no es así. Puede representar, o bien el *nombre* de un archivo particular, o los *nombres* de un conjunto de archivos de un directorio. Si se trata de un conjunto de archivos, se puede preguntar por el conjunto con el método **list( )**, que devuelve un array de **String**s. Tiene sentido devolver un array en vez de una de las clases contenedoras flexibles porque el número de elementos es fijo, y si se desea listar un directorio diferente basta con crear un objeto **File** diferente. De hecho, "FilePath" habría sido un nombre mejor para esta clase. Esta sección muestra un ejemplo de manejo de esta clase, incluyendo la **interfaz FilenameFilter** asociada.

Un generador de listados de directorio

Suponga que se desea ver el contenido de un directorio. El objeto **File** puede listarse de dos formas. Si se llama a **list( )** sin parámetros, se logrará la lista completa de lo que contiene el objeto **File**. Sin embargo, si se desea una lista restringida —por ejemplo, si se desean todos los archivos de extensión **.java**— se usará un "filtro de directorio", que es una clase que indica cómo seleccionar los objetos **File** a mostrar.

He aquí el código para el ejemplo. Nótese que el resultado se ha ordenado sin ningún tipo de esfuerzo, de forma alfabética, usando el método **java.utils.Array.sort( )** y el **ComparadorAlfabetico** definido en el Capítulo 9:

```
//: c11:ListadoDirectorio.java
// Muestra listados de directorios.
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class ListadoDirectorio {
  public static void main(String[] args) {
    File ruta = new File(".");
    String[] lista;
    if(args.length == 0)
      lista = ruta.list();
    else
      lista = ruta.list(new FiltroDirectorio(args[0]));
    Arrays.sort(lista,
      new ComparadorAlfabetico());
    for(int i = 0; i < lista.length; i++)
      System.out.println(lista[i]);
  }
}

class FiltroDirectorio implements FilenameFilter {
  String afn;
  FiltroDirectorio(String afn) { this.afn = afn; }
  public boolean accept(File dir, String name) {
    // Retirar información de ruta:
    String f = new File(name).getName();
    return f.indexOf(afn) != -1;
  }
} ///:~
```

La clase **FiltroDirectorio** "implementa" la **interfaz FilenameFilter**. Es útil ver lo simple que es la **interfaz FilenameFilter**:

```
public interface FilenameFilter {
  boolean accept (File dir, String name);
}
```

Dice que este tipo de objeto proporciona un método denominado **accept( )**. La razón que hay detrás de la creación de esta clase es proporcionar el método **accept( )** al método **list( )** de forma que **list( )** pueda "retrollamar" a **accept( )** para determinar qué nombres deberían ser incluidos en la lista. Por consiguiente, a esta técnica se le suele llamar *retrollamada* o a veces *functor* (es decir, **FiltroDirectorio** es un *functor* porque su único trabajo es albergar un método) o *Patrón Comando*. Dado que **list( )** toma un objeto **FilenameFilter** como parámetro, se le puede pasar un objeto de cualquier clase que implemente **FilenameFilter** para elegir (incluso en tiempo de ejecución) cómo se comportará el método **list( )**. El propósito de una retrollamada es proporcionar flexibilidad al comportamiento del código.

FiltroDirectorio muestra que, justo porque una **interfaz** contenga sólo un conjunto de métodos, uno no está restringido a escribir sólo esos métodos. (Sin embargo, al menos hay que proporcionar definiciones para todos los métodos de la interfaz.) En este caso, se crea también el constructor **FiltroDirectorio**.

El método **accept( )** debe aceptar un objeto **File** que represente el directorio en el que se encuentra un archivo en particular, y un **String** que contenga el nombre de ese archivo. Se podría elegir entre utilizar o ignorar cualquiera de estos parámetros, pero probablemente se usará al menos el nombre del archivo. Debe recordarse que el método **list( )** llama a **accept( )** por cada uno de los nombres de archivo del objeto directorio para ver cuál debería incluirse —lo que se indica por el resultado **boolean** devuelto por **accept( )**.

Para asegurarse de que el elemento con el que se está trabajando es sólo un nombre de archivo sin información de ruta, todo lo que hay que hacer es tomar el **String** y crear un objeto **File** a partir del mismo, después llamar a **getName( )**, que retira toda la información relativa a la ruta (de forma independiente de la plataforma). Después, **accept( )** usa el método **indexOf( )** de la clase **String** para ver si la cadena de caracteres a buscar **afn** aparece en algún lugar del nombre del archivo. Si se encuentra **afn** en el string, el valor devuelto es el índice de comienzo de **afn**, mientras que si no se encuentra, se devuelve el valor -1. Hay que ser conscientes de que es una búsqueda de cadenas de caracteres simple y que no tiene expresiones de emparejamiento de comodines —como por ejemplo "for?.b?*"— lo cual sería más difícil de implementar.

El método **list( )** devuelve un array. Se puede preguntar por la longitud del mismo y recorrerlo seleccionando sus elementos. Esta habilidad de pasar un array hacia y desde un método es una gran mejora frente al comportamiento de C y C++.

Clases internas anónimas

Este ejemplo es ideal para reescribirlo utilizando una clase interna anónima (descritas en el Capítulo 8). En principio, se crea un método **filtrar( )** que devuelve una referencia a **FilenameFilter**:

```
//: c11:ListadoDirectorio2.java
// Usa clases internas anónimas.
import java.io.*;
```

```
import java.util.*;
import com.bruceeckel.util.*;

public class ListadoDirectorio2 {
  public static FilenameFilter
  filtrar(final String afn) {
    // Creación de la clase interna anónima:
    return new FilenameFilter() {
      String fn = afn;
      public boolean accept(File dir, String n) {
        // Retirar información de ruta:
        String f = new File(n).getName();
        return f.indexOf(fn) != -1;
      }
    }; // Fin de la clase interna anónima
  }
  public static void main(String[] args) {
    File ruta = new File(".");
    String[] lista;
    if(args.length == 0)
      lista = ruta.list();
    else
      lista = ruta.list(filtro(args[0]));
    Arrays.sort(lista,
      new ComparadorAlfabetico());
    for(int i = 0; i < lista.length; i++)
      System.out.println(lista[i]);
  }
} ///:~
```

Nótese que el parámetro que se pase a **filtrar( )** debe ser **final**. Esto es necesario para que la clase interna anónima pueda usar un objeto de fuera de su ámbito.

El diseño es una mejora porque la clase **FilenameFilter** está ahora firmemente ligada a **ListadoDirectorio2**. Sin embargo, es posible llevar este enfoque un paso más allá y definir la clase interna anónima como un argumento de **list( )**, en cuyo caso es incluso más pequeña:

```
//: c11:ListadoDirecctorio3.java
// Construyendo la clase interna anónima "en el lugar".
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class ListadoDirectorio3 {
  public static void main(final String[] args) {
    File ruta = new File(".");
    String[] lista;
```

```
      if(args.length == 0)
        lista = ruta.list();
      else
        lista = ruta.list(new FilenameFilter() {
          public boolean
          accept(File dir, String n) {
            String f = new File(n).getName();
            return f.indexOf(args[0]) != -1;
          }
        });
      Arrays.sort(lista,
        new ComparadorAlfabetico());
      for(int i = 0; i < lista.length; i++)
        System.out.println(lista[i]);
    }
  } ///:~
```

El argumento del **main( )** es ahora **final**, puesto que la clase interna anónima usa directamente **args[0]**.

Esto muestra cómo las clases anónimas internas permiten la creación de clases rápida y limpiamente para solucionar problemas. Puesto que todo en Java se soluciona con clases, ésta puede ser una técnica de codificación útil. Un beneficio es que mantiene el código que soluciona un problema en particular aislado y junto en el mismo sitio. Por otro lado, no es siempre fácil de leer, por lo que hay que usarlo juiciosamente.

Comprobando y creando directorios

La clase **File** es más que una simple representación de un archivo o un directorio existentes. También se puede usar un objeto **File** para crear un nuevo directorio o una trayectoria de directorio completa si ésta no existe. También se pueden mirar las características de archivos (tamaño, fecha de la última modificación, lectura/escritura), ver si un objeto **File** representa un archivo o un directorio, y borrar un archivo. Este programa muestra algunos de los otros métodos disponibles con la clase **File** (ver la documentación HTML de *http://java.sun.com* para obtener el conjunto completo):

```
//: c11:CrearDirectorios.java
// Demuestra el uso de la clase File para
// crear directorios y manipular archivos.
import java.io.*;

public class CrearDirectorios {
  private final static String uso =
    "Uso:CrearDirectorios ruta1 ...\n" +
    "Crea cada ruta\n" +
    "Uso:CrearDirectorios -d ruta1 ...\n" +
```

```
      "Borra cada ruta\n" +
      "Uso:CrearDirectorios -r ruta1 ruta2\n" +
      "Renombra ruta1 a ruta2\n";
  private static void uso() {
    System.err.println(uso);
    System.exit(1);
  }
  private static void datosArchivo(File f) {
    System.out.println(
      "Ruta absoluta: " + f.getAbsolutePath() +
      "\n Puede leer: " + f.canRead() +
      "\n Puede escribir: " + f.canWrite() +
      "\n Conseguir el nombre: " + f.getName() +
      "\n Conseguir su padre: " + f.getParent() +
      "\n Conseguir ruta: " + f.getPath() +
      "\n Longitud: " + f.length() +
      "\n Ultima modificacion: " + f.lastModified());
    if(f.isFile())
      System.out.println("Es un archivo");
    else if(f.isDirectory())
      System.out.println("Es un directorio");
  }
  public static void main(String[] args) {
    if(args.length < 1) uso();
    if(args[0].equals("-r")) {
      if(args.length != 3) uso();
      File
        viejo = new File(args[1]),
        nuevo = new File(args[2]);
      viejo.renameTo(nuevo);
      datosArchivo(viejo);
      datosArchivo(nuevo);
      return; // Salir del main
    }
    int contador = 0;
    boolean borrar = false;
    if(args[0].equals("-d")) {
      contador++;
      borrar = true;
    }
    for( ; contador < args.length; contador++) {
      File f = new File(args[contador]);
      if(f.exists()) {
        System.out.println(f + " existe");
        if(borrar) {
```

```
                System.out.println("borrando..." + f);
                f.delete();
            }
        }
        else { // No existe
            if(!borrar) {
                f.mkdirs();
                System.out.println("creado " + f);
            }
        }
        datosArchivo(f);
    }
  }
} ///:~
```

En **datosArchivo( )** se pueden ver varios métodos de investigación que se usan para mostrar la información sobre la trayectoria de un archivo o un directorio.

El primer método ejercitado por el método **main( )** es **renameTo( )**, que permite renombrar (o mover) un archivo a una ruta totalmente nueva representada por un parámetro, que es otro objeto **File**. Esto también funciona con directorios de cualquier longitud.

Si se experimenta con el programa, se verá que se pueden construir rutas de cualquier complejidad, pues **mkdirs( )** hará todo el trabajo.

Entrada y salida

Las bibliotecas de E/S usan a menudo la abstracción del flujo, que representa cualquier fuente o consumidor de datos como un objeto capaz de producir o recibir fragmentos de código. El flujo oculta los detalles de lo que ocurre con los datos en el dispositivo de E/S real.

Las clases de E/S de la biblioteca de Java se dividen en entrada y salida, como ocurre con los datos en el dispositivo de E/S real. Por herencia, todo lo que deriva de las clases **InputStream** o **Reader** tiene los métodos básicos **read( )** para leer un simple byte o un array de bytes. Asimismo, todo lo que derive de las clases **OutputStream** o **Writer** tiene métodos básicos denominados **write( )** para escribir un único byte o un array de bytes. Sin embargo, generalmente no se usarán estos métodos; existen para que otras clases puedan utilizarlos —estas otras clases proporcionan una interfaz más útil. Por consiguiente, rara vez se creará un objeto flujo usando una única clase, sino que se irán apilando en capas diversos objetos para proporcionar la funcionalidad deseada. El hecho de crear más de un objeto para crear un flujo resultante único es la razón primaria por la que la biblioteca de flujos de Java es tan confusa.

Ayuda bastante clasificar en tipos las clases en base a su funcionalidad. En Java 1.0, los diseñadores de bibliotecas comenzaron decidiendo que todas las clases relacionadas con entrada heredarían de **InputStream**, y que todas las asociadas con la salida heredarían de **OutputStream**.

Tipos de **InputStream**

El trabajo de **InputStream** es representar las clases que producen entradas desde distintas fuentes. Éstas pueden ser:

1. Un array de bytes.
2. Un objeto **String**.
3. Un archivo.
4. Una "tubería", que funciona como una tubería física: se ponen elementos en un extremo y salen por el otro.
5. Una secuencia de otros flujos, de forma que se puedan agrupar todos en un único flujo.
6. Otras fuentes, como una conexión a Internet (esto se verá al final del capítulo).

Cada una de éstas tiene asociada una subclase de **InputStream**. Además, el **FilterInputStream** es también un tipo de **InputStream**, para proporcionar una clase base para las clases "decoradoras" que adjuntan atributos o interfaces útiles a los flujos de entrada. Esto se discutirá más tarde.

Tabla 11-1. Tipos de InputStream

Clase	Función	Parámetros del constructor
		Cómo usarla
ByteArray-InputStream	Permite usar un espacio de almacenamiento intermedio de memoria como un **InputStream**	El intermedio del que extraer los bytes. Espacio de almacenamiento.
		Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar un interfaz útil.
StringBuffer-InputStream	Convierte un **String** en un **InputStream**	Un **String**. La implementación subyacente usa, de hecho, un **StringBuffer**.
		Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
File-InputStream	Para leer información de un archivo	Un **String** que represente al nombre del archivo, o un objeto **File** o **FileDescriptor**.
		Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.

Clase	Función	Parámetros del constructor Cómo usarla
Piped-InputStream	Produce los datos que se están escribiendo en el **PipedOutputStream** asociado. Implementa el concepto de "entubar".	**PipedOutputStream.** Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
Sequence-InputStream	Convierte dos o más objetos **InputStream** en un **InputStream** único.	Dos objetos **InputStream** o una **Enumeration** para contener objetos **InputStream.** Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
Filter-InputStream	Clase abstracta que es una interfaz para los decoradores que proporcionan funcionalidad útil a otras clases **InputStream.** Ver Tabla 11-3.	Ver Tabla 11-3. Ver Tabla 11-3.

Tipos de **OutputStream**

Esta categoría incluye las clases que deciden dónde irá la salida: un array de bytes (sin embargo, no **String**; presumiblemente se puede crear uno usando un array de bytes), un fichero, o una "tubería".

Además, el **FilterOutputStream** proporciona una clase base para las clases "decorador" que adjuntan atributos o interfaces útiles de flujos de salida. Esto se discute más tarde.

Tabla 11-2. Tipos de OutputStream

Clase	Función	Parámetros del constructor Cómo usarla
ByteArray-OutputStream	Crea un espacio de almacenamiento intermedio en memoria. Todos los datos que se envían al flujo se ubican en este espacio de almacenamiento intermedio.	Tamaño opcional inicial del espacio de almacenamiento intermedio. Para designar el destino de los datos. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.

Clase	Función	Parámetros del constructor Cómo usarla
File-OutputStream	Para enviar información a un archivo.	Un **String**. que representa el nombre de archivo, o un objeto **File** o un objeto **FileDescriptor.** Para designar el destino de los datos. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.
Piped-OutputStream	Cualquier información que se desee escribir aquí acaba automáticamente como entrada del **PipedInputStream** asociado. Implementa el concepto de "entubar".	**PipedInputStream** Para designar el destino de los datos para multihilo. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.
Filter-OutputStream	Clase abstracta que es una interfaz para los decoradores que proporcionan funcionalidad útil a las otras clases **OutputStream.** Ver Tabla 11-4.	Ver Tabla 11-4. Ver Tabla 11-4.

Añadir atributos e interfaces útiles

Al uso de objetos en capas para añadir dinámica y transparentemente responsabilidades a objetos individuales se le denomina patrón *Decorador.* (Los Patrones[1] son el tema central de *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com.*) El patrón decorador especifica que todos los objetos que envuelvan el objeto inicial tengan la misma interfaz. Así se hace uso de la transparencia del decorador —se envía el mismo mensaje a un objeto esté o no decorado. Éste es el motivo de la existencia de clases "filtro" en la biblioteca E/S de Java: la clase abstracta "filter" es la clase base de todos los decoradores. (Un decorador debe tener la misma interfaz que el objeto que decora, pero el decorador también puede extender la interfaz, lo que ocurre en muchas de las clases "filter".)

Los decoradores se han usado a menudo cuando se generan numerosas subclases para satisfacer todas las combinaciones posibles necesarias —tantas clases que resultan poco prácticas. La biblio-

[1] *Desing Patterns,* Erich Gamma *et al.*, Addison-Wesley 1995.

teca de E/S de Java requiere muchas combinaciones distintas de características, siendo ésta la razón del uso del patrón decorador. Sin embargo este patrón tiene un inconveniente. Los decoradores dan mucha más flexibilidad al escribir un programa (puesto que se pueden mezclar y emparejar atributos fácilmente), pero añaden complejidad al código. La razón por la que la biblioteca de E/S es complicada de usar es que hay que crear muchas clases —los tipos de E/S básicos más todos los decoradores— para lograr el único objeto de E/S que se quiere.

Las clases que proporcionan la interfaz decorador para controlar un **InputStream** o un **Output Stream** particular son **FilertInputStream** y **FilterOuputStream** —que no tienen nombres muy intuitivos. Éstas dos últimas clases son abstractas, derivadas de las clases base de la biblioteca de E/S, **InputStream** y **OutputStream**, el requisito clave del decorador (de forma que proporciona la interfaz común a todos los objetos que se están decorando).

Leer de un **InputStream** con un **FilterInputStream**

Las clases **FilterInputStream** llevan a cabo dos cosas significativamente diferentes. **DataInput Stream** permite leer distintos tipos de datos primitivos, además de objetos **String**. (Todos los métodos empiezan por "read", como **readByte( )**, **readFloat( )**, etc.) Esto, junto con su compañero **DataOutputStream**, permite mover datos primitivos de un sitio a otro vía un flujo. Estos "lugares" vendrán determinados por las clases de la Tabla 11-1.

Las clases restantes modifican la forma de comportarse internamente de **InputStream**: haga uso o no de espacio de almacenamiento intermedio, si mantiene un seguimiento de las líneas que lee (permitiendo preguntar por el número de líneas, o establecerlo) o si se puede introducir un único carácter. Las dos últimas clases se parecen mucho al soporte para la construcción de un compilador (es decir, se añadieron para poder construir el propio compilador de Java), por lo que probablemente no se usen en programación en general.

Probablemente se necesitará pasar la entrada por un espacio de almacenamiento intermedio casi siempre, independientemente del dispositivo de E/S al que se esté conectado, por lo que tendría más sentido que la biblioteca de E/S fuera un caso excepcional (o simplemente una llamada a un método) de entrada sin espacio de almacenamiento intermedio, en vez de una entrada con espacio de almacenamiento intermedio.

Tabla 11-3. Tipos de FilterInputStream

Clase	Función	Parámetros del constructor Cómo usarla
Data-InputStream	Usado junto con **DataOutputStream**, de forma que se puedan leer datos primitivos (**int**, **char**, **long**, etc.) de un flujo de forma portable.	**InputStream** Contiene una interfaz completa que permite leer tipos primitivos.

Clase	Función	Parámetros del constructor Cómo usarla
Buffered-InputStream	Se usa para evitar una lectura cada vez que se soliciten nuevos datos. Se está diciendo "utiliza un espacio de almacenamiento intermedio".	**InputStream**, con tamaño de espacio de almacenamiento intermedio opcional. No proporciona una interfaz *per se*, simplemente el requisito de que se use un espacio de almacenamiento intermedio. Adjuntar un objeto interfaz.
LineNumber-InputStream	Mantiene un seguimiento de los números de línea en el *flujo* de entrada; se puede llamar a **getLineNumber( )** y a **setLineNumber(int)**.	**InputStream** Simplemente añade numeración de líneas, por lo que probablemente se adjunte un objeto interfaz.
Pushback-InputStream	Tiene un espacio de almacenamiento intermedio de un byte para el último carácter a leer.	**InputStream.** Se usa generalmente en el escáner de un compilador y probablemente se incluyó porque lo necesitaba el compilador de Java. Probablemente prácticamente nadie la utilice.

Escribir en un **OutputStream** con **FilterOutputStream**

El complemento a **DataInputStream** es **DataOutputStream**, que da formato a los tipos primitivos y objetos **String**, convirtiéndolos en un flujo de forma que cualquier **DataInputStream**, de cualquier máquina, los pueda leer. Todos los métodos empiezan por "write", como **writeByte( )**, **writeFloat( )**, etc.

La intención original para **PrintStream** era que imprimiera todos los tipos de datos primitivos así como los objetos **String** en un formato visible. Esto es diferente de **DataOutputStream**, cuya meta es poner elementos de datos en un flujo de forma que **DataInputStream** pueda reconstruirlos de forma portable.

Los dos métodos importantes de **PrintStream** son **print( )** y **println( )**, sobrecargados para imprimir todo los tipos. La diferencia entre **print( )** y **println( )** es que la útlima añade una nueva línea al acabar.

PrintStream puede ser problemático porque captura todas las **IOExpections**. (Hay que probar explícitamente el estado de error con **checkError( )**, que devuelve **true** si se ha producido algún error.) Además, **PrintStream** no se internacionaliza adecuadamente y no maneja saltos de línea independientemente de la plataforma (estos problemas se solucionan con **PrintWriter**).

BufferedOutputStream es un modificador que dice al flujo que use espacios de almacenamiento intermedio, de forma que no se realice una lectura física cada vez que se escribe en el flujo. Probablemente siempre se deseará usarlo con archivos, y probablemente la E/S de consola.

Tabla 11-4. Tipos de FilterOutputStream

Clase	Función	Parámetros del constructor Cómo usarla
Data-OutputStream	Usado junto con **DataInputStream**, de forma que se puedan escribir datos primitivos (int, char, long, etc.) de un flujo de forma portable.	**OutputStream** Contiene una interfaz completa que permite escribir tipos de datos primitivos.
PrintStream	Para producir salida formateada. Mientras que **DataOutputStream** maneja el *almacenamiento* de datos, **PrintStream** maneja su *visualización*.	**OutputStream** con un boolean opcional que indica que se vacía el espacio de almacenamiento intermedio con cada nueva línea. Debería ser la envoltura "final" del objeto **OutputStream.** Probablemente se usará mucho.
Buffered-OutputStream	Se usa para evitar una escritura física cada vez que se envía un fragmento de datos. Se está diciendo "Usar un espacio de almacenamiento intermedio". Se puede llamar a **flush( )** para vaciar el espacio de almacenamiento intermedio.	**OutputStream** con tamaño del espacio de almacenamiento intermedio. No proporciona una interfaz *per se*, simplemente pide que se use un espacio de almacenamiento intermedio. Adjuntar un objeto interfaz.

Readers & Writers

Java 1.1. hizo algunas modificaciones fundamentales a la biblioteca de flujos de E/S fundamental de Java (sin embargo, Java 2 no aportó modificaciones significativas). Cuando se observan las clases **Reader** y **Writer**, en un principio se piensa (como hicimos) que su intención es reemplazar las clases **InputStream** y **OutputStream**. Pero ése no es el caso. Aunque se desecharon algunos aspectos de la biblioteca de flujos original (si se usan estos aspectos se recibirá un aviso del compilador), las clases **InputStream** y **OutputStream** siguen proporcionando una funcionalidad valiosa en la forma de E/S orientada al **byte**, mientras que las clases **Reader** y **Writer** proporcionan E/S compatible Unicode basada en caracteres. Además:

1. Java 1.1 añadió nuevas clases a la jerarquía **InputStream** y **OutputStream**, por lo que es obvio que no se estaban reemplazando estas clases.
2. Hay ocasiones en las que hay que usar clases de la jerarquía "byte" *en combinación con* clases de la jerarquía "carácter". Para lograr esto hay clases "puente": **InputStreamReader** convierte un **InputStream** en un **Reader**, y **OutputStreamWriter** convierte un **OutputStream** en un **Writer**.

La razón más importante para la existencia de las jerarquías **Reader** y **Writer** es la internacionalización. La antigua jerarquía de flujos de E/S sólo soporta flujos de 8 bits no manejando caracteres Unicode de 16 bits correctamente. Puesto que Unicode se usa con fines de internacionalización (y el **char** nativo de Java es Unicode de 16 bits), se añadieron las jerarquías **Reader** y **Writer** para dar soporte Unicode en todas las operaciones de E/S. Además, se diseñaron las nuevas bibliotecas de forma que las operaciones se llevaran a cabo de forma más rápida que antiguamente.

En la práctica y en este libro, intentaremos proporcionar un repaso de las clases, pero asumimos que se usará la documentación *en línea* para concretar todos los detalles, así como una lista exhaustiva de los métodos.

Fuentes y consumidores de datos

Casi todas las clases de flujos de E/S de Java originales tienen sus correspondientes clases **Reader** y **Writer** para proporcionar manipulación Unicode nativa. Sin embargo, hay algunos lugares en los que la solución correcta la constituyen los **InputStreams** y **OutputStreams** orientados a **byte**; concretamente, las bibliotecas **java.util.zip** son orientadas a **byte** en vez de orientadas a **char**. Por tanto, el enfoque más sensato es *intentar* usar las clases **Reader** y **Writer** siempre que se pueda, y se descubrirán posteriormente aquellas situaciones en las que hay que usar las otras bibliotecas, ya que el código no compilará.

He aquí una tabla que muestra la correspondencia entre fuentes y consumidores de información (es decir, de dónde y a dónde van físicamente los datos) dentro de ambas jeraquías:

Fuentes & Cusumidores: Clase Java 1.0	**Clase Java 1.1 correspondiente**
InputStream	**Reader** convertidor: **InputStreamReader**
OutputStream	**Writer** convertidor: **OutputStreamWriter**

FileInputStream	**FileReader**
FileOutputStream	**FileWriter**
StringBufferInputStream	**StringReader**
(sin clase correspondiente)	**StringWriter**
ByteArrayInputStream	**CharArrayReader**
ByteArrayOutputStream	**CharArrayWriter**
PipedInputStream	**PipedReader**
PipedOutputStream	**PipedWriter**

En general, se descubrirá que las interfaces de ambas jerarquías son similares cuando no idénticas.

Modificar el comportamiento del flujo

En el caso de **InputStreams** y **OutputStreams** se adaptaron los flujos para necesidades particulares utilizando subclases "decorador" de **FilterInputStream** y **FilterOutputStream**. Las jerarquías de clases **Reader** y **Writer** continúan usando esta idea —aunque no exactamente.

En la tabla siguiente, la correspondencia es una aproximación más complicada que en la tabla anterior. La diferencia se debe a la organización de las clases: mientras que **BufferedOutputStream** es una subclase de **FilterOuputStream**, **BufferedWriter** *no* es una subclase de **FilterWriter** (que, aunque es **abstract**, no tiene subclases, por lo que parece haberse incluido como contenedor o simplemente de forma que nadie la busque sin fruto). Sin embargo, las interfaces de las clases coinciden bastante:

Filtros **Clase Java 1.0**	**Clase Java 1.1 correspondiente**
FilterInputStream	**FilterReader**
FilterOutputStream	**FilterWriter** (clase abstracta sin subclases)
BufferedInputStream	**BufferedReader** (también tiene **readLine( )**)
BufferedOutputStream	**BufferedWriter**
DataInputStream	Usar **DataInputStream** (Excepto cuando se necesite usar **readLine( )**, caso en que debería usarse un **BufferedReader**)
PrintStream	**PrintWriter**

LineNumberInputStream	**LineNumberReader**
StreamTokenizer	**StreamTokenizer** (usar en vez de ello el constructor que toma un **Reader**)
PushBackInputStream	**PushBackReader**

Hay algo bastante claro: siempre que se quiera usar un **readLine( )**, no se debería hacer con un **DataInputStream** nunca más (se mostrará en tiempo de compilación un mensaje indicando que se trata de algo obsoleto), sino que debe usarse en su lugar un **BufferedReader**. **DataInputStream**, sigue siendo un miembro "preferente" de la biblioteca de E/S.

Para facilitar la transición de cara a usar un **PrintWriter**, éste tiene constructores que toman cualquier objeto **OutputStream**, además de objetos **Writer**. Sin embargo, **PrintWriter** no tiene más soporte para dar formato que el proporcionado por **PrintStream**; las interfaces son prácticamente las mismas.

El constructor **PrintWriter** también tiene la opción de hacer vaciado automático, lo que ocurre tras todo **println( )** si se ha puesto a uno el *flag* del constructor.

Clases no cambiadas

Java 1.1 no cambió algunas clases de Java 1.0:

Clases de Java 1.0 sin clases correspondientes Java 1.1
DataOutputStream
File
RandomAccessFile
SequenceInputStream

DataOutputStream, en particular, se usa sin cambios, por tanto, para almacenar y recuperar datos en un formato transportable se usan las jerarquías **InputStream** y **OutputStream**.

Por sí mismo: RandomAccessFile

RandomAccessFile se usa para los archivos que contengan registros de tamaño conocido, de forma que se puede mover de un registro a otro utilizando **seek( )**, para después leer o modificar los registros. Éstos no tienen por qué ser todos del mismo tamaño; simplemente hay que poder determinar lo grandes que son y dónde están ubicados dentro del archivo.

En primera instancia, cuesta creer que **RandomAccessFile** no es parte de la jeraquía **InputStream** o **OutputStream**. Sin embargo, no tiene ningún tipo de relación con esas jerarquías con la excepción de que implementa las interfaces **DataInput** y **DataOutput** (que también están implementados por **DataInputStream** y **DataOutputStream**). Incluso no usa ninguna funcionalidad de las clases **InputStream** u **OutputStream** existentes —es una clase totalmente independiente, escrita de la nada, con métodos exclusivos (en su mayoría nativos). La razón para ello puede ser que **RandomAccessFile** tiene un comportamiento esencialmente distinto al de otros tipos de E/S, puesto que se puede avanzar y retroceder dentro de un archivo. Permanece aislado, como un descendiente directo de **Object**.

Fundamentalmente, un **RandomAccessFile** funciona igual que un **DataInputStream** unido a un **DataOutputStream**, junto con los métodos **getFilePointer( )** para averiguar la posición actual en el archivo, **seek( )** para moverse a un nuevo punto del archivo, y **length( )** para determinar el tamaño máximo del mismo. Además, los constructores requieren de un segundo parámetro (idéntico al de **fopen( )** en C) que indique si se está simplemente leyendo ("**r**") al azar, o leyendo y escribiendo ("**rw**"). No hay soporte para archivos de sólo escritura, lo cual podría sugerir que **RandomAccessFile** podría haber funcionado bien si hubiera heredado de **DataInputStream**.

Los métodos de búsqueda sólo están disponibles en **RandomAccessFile**, que sólo funciona para archivos. **BufferedInputStream** permite **mark( )** una posición (cuyo valor se mantiene en una variable interna única) y hacer un **reset( )** a esa posición, pero no deja de ser limitado y, por tanto, no muy útil.

Usos típicos de flujos de E/S

Aunque se pueden combinar las clases *de flujos* de E/S de muchas formas, probablemente cada uno sólo haga uso de unas pocas combinaciones. Se puede usar el siguiente ejemplo como una referencia básica; muestra la creación y uso de las configuraciones de E/S típicas. Nótese que cada configuración empieza con un número comentado y un título que corresponde a la cabecera de la explicación apropiada que sigue en el texto.

```
//: c11:DemoFlujoES.java
// Configuraciones típicas de flujos de E/S.
import java.io.*;

public class DemoFlujoES {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    // 1. Leyendo de la entrada línea a línea:
    BufferedReader entrada =
      new BufferedReader(
        new FileReader("DemoflujoES.java"));
    String S, S2 = new String();
    while((S = entrada.readLine())!= null)
```

```
    S2 += S + "\n";
  entrada.close();

  // 1b. Leyendo de la entrada estándar:
  BufferedReader entradaEstandar =
    new BufferedReader(
      new InputStreamReader(System.in));
  System.out.print("Introduce una linea:");
  System.out.println(entradaEstandar.readLine());

  // 2. Entrada desde memoria
  StringReader entrada2 = new StringReader(S2);
  int c;
  while((c = entrada2.read()) != -1)
    System.out.print((char)c);

  // 3. Entada con formato desde memoria
  try {
    DataInputStream entrada3 =
      new DataInputStream(
        new ByteArrayInputStream(s2.getBytes()));
    while(true)
      System.out.print((char)entrada3.readByte());
  } catch(EOFException e) {
    System.err.println("Fin del flujo");
  }

  // 4. Salida de archivo
  try {
    BufferedReader entrada4 =
      new BufferedReader(
        new StringReader(s2));
    PrintWriter salida1 =
      new PrintWriter(
        new BufferedWriter(
          new FileWriter("DemoES.out")));
    int contadorLineas = 1;
    while((s = entrada4.readLine()) != null )
      salida1.println(contadorLineas++ + ": " + s);
    salida1.close();
  } catch(EOFException e) {
    System.err.println("Fin del flujo");
  }

  // 5. Almacenando & recuperando datos
```

```
try {
   DataOutputStream salida2 =
      new DataOutputStream(
         new BufferedOutputStream(
            new FileOutputStream("Datos.txt")));
   salida2.writeDouble(3.14159);
   salida2.writeChars("Eso era pi\n");
   salida2.writeBytes("Eso era pi\n");
   salida2.close();
   DataInputStream entrada5 =
      new DataInputStream(
         new BufferedInputStream(
            new FileInputStream("Datos.txt")));
   BufferedReader entrada5br =
      new BufferedReader(
         new InputStreamReader(entrada5));
   // Hay que usar DataInputStream para datos:
   System.out.println(entrada5.readDouble());
   // Ahora se puede usar el readLine() "apropiado":
   System.out.println(entrada5br.readLine());
   // Pero la línea resulta divertida.
   // La creada con writeBytes es correcta:
   System.out.println(entrada5br.readLine());
} catch(EOFException e) {
   System.err.println("Fin del flujo");
}

// 6. Leyendo/escribiendo archivos de acceso directo
RandomAccessFile rf =
   new RandomAccessFile("rprueba.dat", "rw");
for(int i = 0; i < 10; i++)
   rf.writeDouble(i*1.414);
rf.close();

rf =
   new RandomAccessFile("rprueba.dat", "rw");
rf.seek(5*8);
rf.writeDouble(47.0001);
rf.close();

rf =
   new RandomAccessFile("rprueba.dat", "r");
for(int i = 0; i < 10; i++)
   System.out.println(
      "Valor " + i + ": " +
```

```
            rf.readDouble());
        rf.close();
    }
} ///:~
```

He aquí las descripciones de las secciones numeradas del programa:

Flujos de entrada

Las Secciones 1-4 demuestran la creación y uso de flujos de entrada. La Sección 4 muestra también el uso simple de un flujo de salida.

1. Archivo de entrada utilizando espacio de almacenamiento intermedio

Para abrir un archivo para entrada de caracteres se usa un **FileInputReader** junto con un objeto **File** o **String** como nombre de archivo. Para lograr mayor velocidad, se deseará que el archivo tenga un espacio de almacenamiento intermedio de forma que se dé la referencia resultante al constructor para un **BufferedReader**. Dado que **BufferedReader** también proporciona el método **readLine( )**, éste es el objeto final y la interfaz de la que se lee. Cuando se llegue al final del archivo, **readLine( )** devuelve **null**, por lo que es éste el valor que se usa para salir del bucle **while**.

El **String s2** se usa para acumular todo el contenido del archivo (incluyendo las nuevas líneas que hay que añadir porque **readLine( )** las quita). Después se usa **s2** en el resto de porciones del programa. Finalmente, se invoca a **close( )** para cerrar el archivo. Técnicamente, se llamará a **close( )** cuando se ejecute **finalize( )**, cosa que se supone que ocurrirá (se active o no el recolector de basura) cuando se acabe el programa. Sin embargo, esto se ha implementado inconsistentemente, por lo que el único enfoque seguro es el de invocar explícitamente a **close( )** en el caso de manipular archivos.

La sección 1b muestra cómo se puede envolver **System.in** para leer la entrada de la consola. **System.in** es un **DataInputStream** y **BufferedReader** necesita un parámetro **Reader**, por lo que se hace uso de **InputStreamReader** para llevar a cabo la traducción.

2. Entrada desde memoria

Esta sección toma el **String s2**, que ahora contiene todos los contenidos del archivo y lo usa para crear un **StringReader**. Después se usa **read( )** para leer cada carácter de uno en uno y enviarlo a la consola. Nótese que **read( )** devuelve el siguiente byte como un **int** por lo que hay que convertirlo en **char** para que se imprima correctamente.

3. Entrada con formato desde memoria

Para ler datos "con formato", se usa un **DataInputStream**, que es una clase de E/S orientada a **byte** (en vez de orientada a **char**). Por consiguiente se deben usar todas las clases **InputStream** en vez de clases **Reader**. Por supuesto, se puede leer cualquier cosa (como un archivo) como si de bytes se tratara, usando clases **InputStream**, pero aquí se usa un **String**. Para convertir el **String** en un array de

bytes, apropiado para un **ByteArrayInputStream, String** tiene un método **getBytes( )** que se encarga de esto. En este momento, se tiene un **InputStream** apropiado para manejar **DataInputStream**.

Si se leen los caracteres de un **DataInputStream** de uno en uno utilizando **readByte( )**, cualquier valor de byte constituye un resultado legítimo por lo que no se puede usar el valor de retorno para detectar el final de la entrada. En vez de ello, se puede usar el método **available( )** para averiguar cuántos caracteres más quedan disponibles. He aquí un ejemplo que muestra cómo leer un archivo byte a byte:

```
//: c11:PruebaEOF.java
// Probando el fin de archivo
// al leer de byte en byte.
import java.io.*;

public class PruebaEOF {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    DataInputStream entrada =
      new DataInputStream(
       new BufferedInputStream(
        new FileInputStream("PruebaEOF.java")));
    while(entrada.available() != 0)
      System.out.print((char)entrada.readByte());
  }
} ///:~
```

Nótese que **available( )** funciona de forma distinta en función del tipo de medio desde el que se esté leyendo; literalmente es "el número de bytes que se pueden leer *sin bloqueo*". Con archivos, esto equivale a todo el archivo pero con un tipo de flujo distinto podría no ser así, por lo que debe usarse con mucho cuidado.

También se podría detectar el fin de la entrada en este tipo de casos mediante una excepción. Sin embargo, el uso de excepciones para flujos de control se considera un mal uso de esta característica.

4. Salida a archivo

Este ejemplo también muestra cómo escribir datos en un archivo. En primer lugar, se crea un **FileWriter** para conectar con el archivo. Generalmente se deseará pasar la salida a través de un espacio de almacenamiento intermedio, por lo que se genera un **BufferedWriter** (es conveniente intentar retirar este envoltorio para ver su impacto en el rendimiento —el uso de espacios de almacenamiento intermedio tiende a incrementar considerablemente el rendimiento de las operaciones de E/S). Después, se convierte en un **PrintWriter** para hacer uso de las opciones de dar formato. El archivo de datos que se cree así es legible como un archivo de texto normal y corriente.

A medida que se escriban líneas al archivo, se añaden los números de línea. Nótese que no se *usa* **LineNumberInputStream**, porque es una clase estúpida e innecesaria. Como se muestra en este caso, es fundamental llevar a cabo un seguimiento de los números de página.

Cuando se agota el flujo de entrada, **readLine( )** devuelve **null**. Se verá una llamada **close( )** explícita para **salida1**, porque si no se invoca a **close( )** para todos los archivos de salida, los espacios de almacenamiento intermedio no se vaciarán, de forma que las operaciones pueden quedar inacabadas.

Flujos de salida

Los dos tipos primarios de flujos de salida se diferencian en la forma de escribir los datos: uno lo hace de forma comprensible para el ser humano, y el otro lo hace para pasárselos a **DataInput Stream**. El **RandomAccessFile** se mantiene independiente, aunque su formato de datos es compatible con **DataInputStream** y **DataOutputStream**.

5. Almacenar y recuperar datos

Un **PrintWriter** da formato a los datos de forma que sean legibles por el hombre. Sin embargo, para sacar datos de manera que puedan ser recuperados por otro flujo, se usa un **DataOutputStream para escribir los datos** y un **DataInputStream** para la recuperación. Por supuesto, estos flujos podrían ser cualquier cosa, pero aquí se usa un archivo con espacios de almacenamiento intermedio tanto para la lectura como para la escritura. **DataOutputStream** y **DataInputStream** están orientados a **byte**, por lo que requieren de **InputStreams** y **OutputStreams**.

Si se usa un **DataOutputStream** para escribir los datos, Java garantiza que se pueda recuperar el dato utilizando eficientemente un **DataInputStream** —independientemente de las plataformas sobre las que se lleven a cabo las operaciones de lectura y escritura. Esto tiene un valor increíble, pues nadie sabe quién ha invertido su tiempo preocupándose por aspectos de datos específicos de cada plataforma. El problema se desvanece simplemente teniendo Java en ambas plataformas[2].

Nótese que se escribe la cadena de caracteres haciendo uso tanto de **writeChars( )** como de **writeBytes( )**. Cuando se ejecute el programa, se observará que **writeChars( )** saca caracteres Unicode de 16 bits. Cuando se lee la línea haciendo uso de **readLine( )** se verá que hay un espacio entre cada carácter, que es debido al byte extra introducido por Unicode. Puesto que no hay ningún método "readChars" complementario en **DataInputStream**, no queda más remedio que sacar esos caracteres de uno en uno con **readChar( )**. Por tanto, en el caso de ASCII, es más sencillo escribir los caracteres como bytes seguidos de un salto de línea; posteriormente se usa **readLine( )** para leer de nuevo esos bytes en una línea ASCII tradicional.

El **writeDouble( )** almacena el número **double** en el flujo y el **readDouble( )** complementario lo recupera (hay métodos semejantes para hacer lo mismo en la escritura y lectura de otros tipos). Pero para que cualquiera de estos métodos de lectura funcione correctamente es necesario conocer la ubicación exacta del elemento de datos dentro del flujo, puesto que sería igualmente posible

---

[2] XML es otra solución al mismo problema: mover datos entre plataformas de computación diferentes, que en este caso no depende de que haya Java en ambas plataformas. Además, existen herramientas Java para dar soporte a XML.

leer el **double** almacenado como una simple secuencia de bytes, o como un **char**, etc. Por tanto, o bien hay que establecer un formato fijo para los datos dentro del archivo, o hay que almacenar en el propio archivo información extra que será necesario analizar para determinar dónde se encuentra ubicado el dato.

6. Leer y escribir archivos de acceso aleatorio

Como se vio anteriormente, el **RandomAccessFile** se encuentra casi totalmente aislado del resto de la jerarquía de E/S, excepto por el hecho de que implementa las interfaces **DataInput** y **DataOutput**. Por tanto, no se puede combinar con ninguno de los aspectos de las subclases **InputStream** y **OutputStream**. Incluso aunque podría tener sentido tratar un **ByteArrayInputStream** como un elemento de acceso aleatorio, se puede usar **RandomAccessFile** simplemente para abrir un archivo. Hay que asumir que un **RandomAccessFile** tiene sus espacios de almacenamiento intermedio, así que no hay que añadírselos.

La opción que queda es el segundo parámetro del constructor: se puede abrir un **RandomAccessFile** para leer (**"r"**), o para leer y escribir (**"rw"**).

La utilización de un **RandomAccessFile** es como usar un **DataInputStream** y un **DataOutput Stream** combinados (puesto que implementa las interfaces equivalentes). Además, se puede ver que se usa **seek( )** para moverse por el archivo y cambiar algunos de sus valores.

¿Un error?

Si se echa un vistazo a la Sección 6, se verá que el dato se escribe *antes* que el texto. Esto se debe a un problema que se introdujo con Java 1.1 (y que persiste en Java 2) que parece un error, pero informamos de él y la gente de JavaSoft que trabaja en errores nos informó de que funciona exactamente como se desea que funcione (sin embargo, el problema *no* ocurría en Java 1.0, y eso nos hace sospechar). El problema se muestra en el ejemplo siguiente:

```
//: c11:ProblemaES.java
// Problema de E/S de Java 1.1 y superiores.
import java.io.*;

public class ProblemaES {
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    DataOutputStream salida =
      new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream("Datos.txt")));
    salida.writeDouble(3.14159);
    salida.writeBytes("Este es el valor de pi\n");
    salida.writeBytes("El valor de pi/2 es:\n");
    salida.writeDouble(3.14159/2);
```

```
      salida.close();

      DataInputStream entrada =
        new DataInputStream(
          new BufferedInputStream(
            new FileInputStream("Datos.txt")));
      BufferedReader entradabr =
        new BufferedReader(
          new InputStreamReader(entrada));
      // Se escriben los doubles ANTES de la lectura correcta
      // de la línea de texto:
      System.out.println(entrada.readDouble());
      // Leer las líneas de texto:
      System.out.println(entradabr.readLine());
      System.out.println(entradabr.readLine());
      // Intentar leer los doubles después de la línea
      // produce una excepción de fin-de-fichero:
      System.out.println(entrada.readDouble());
  }
} ///:~
```

Parece que todo lo que se escribe tras una llamada a **writeBytes( )** no es recuperable. La respuesta es aparentemente la misma que en el viejo chiste: "Doctor, ¡cuando hago esto, me duele!" "¡Pues no lo haga!"

Flujos entubados

PipedInputStream, **PipedOutputStream**, **PipedReader** y **PipedWriter** ya se han mencionado anteriormente en este capítulo, aunque sea brevemente. No se pretende, no obstante, sugerir que no sean útiles, pero es difícil descubrir su verdadero valor hasta entender el multihilo, puesto que este tipo de flujos se usa para la comunicación entre hilos. Su uso se verá en un ejemplo del Capítulo 14.

E/S estándar

El término *E/S estándar* proviene de Unix (si bien se ha reproducido tanto en Windows como en otros muchos sistemas operativos). Hace referencia al flujo de información que utiliza todo programa. Así, toda la entrada a un programa proviene de la *entrada estándar*, y su salida "fluye" a través de la *salida estándar*, mientras que todos sus mensajes de error se envían a la *salida de error estándar*. El valor de la E/S estándar radica en la posibilidad de encadenar estos programas de forma sencilla de manera que la salida estándar de uno se convierta en la entrada estándar del siguiente. Esta herramienta resulta extremadamente poderosa.

Leer de la entrada estándar

Siguiendo el modelo de E/S estándar, Java tiene **Sytem.in**, **System.out** y **System.err**. A lo largo de todo este libro, se ha visto cómo escribir en la salida estándar haciendo uso de **System.out**, que ya viene envuelto como un objeto **PrintStream**. **System.err** es igual que un **PrintStream**, pero **System.in** es como un **InputStream** puro, sin envoltorios. Esto significa que, si bien se pueden utilizar **System.out** y **System.err** directamente, es necesario envolver de alguna forma **System.in** antes de poder leer de él.

Generalmente se desea leer una entrada línea a línea haciendo uso de **readLine( )**, por lo que se deseará envolver **System.in** en un **BufferedReader**. Para ello hay que convertir **System.in** en un **Reader** haciendo uso de **InputStreamReader**. He aquí un ejemplo que simplemente visualiza toda línea que se teclee:

```
//: c11:Eco.java
// Como leer de la entrada estándar.
import java.io.*;

public class Eco {
  public static void main(String[] args)
  throws IOException {
    BufferedReader entrada =
        new BufferedReader(
          new InputStreamReader(System.in));
    String s;
    while((s = entrada.readLine()).length() != 0)
      System.out.println(s);
    // El programa acaba con una línea vacía.
  }
} ///:~
```

La razón que justifica la especificación de la excepción es que **readLine( )** puede lanzar una **IOException**. Nótese que **System.in** debería utilizar un espacio de almacenamiento intermedio, al igual que la mayoría de flujos.

Convirtiendo **System.out** en un **PrintWriter**

System.out es un **PrintStream**, que es, a su vez, un **OutputStream**. **PrintWriter** tiene un constructor que toma un **OutputStream** como parámetro. Por ello, si se desea es posible convertir **System.out** en un **PrintWriter** haciendo uso de ese constructor:

```
//: c11:CambiarSistemOut.java
// Convertir System.out en un PrintWriter.
import java.io.*;

public class CambiarSistemOut {
  public static void main(String[] args) {
```

```
      PrintWriter salida =
        new PrintWriter(System.out, true);
      salida.println("Hola, mundo");
    }
} ///:~
```

Es importante usar la versión de dos parámetros del constructor **PrintWriter** y poner el segundo parámetro a **true** para habilitar el vaciado automático, pues, si no, puede que no se vea la salida.

Redirigiendo la E/S estándar

La clase **System** de Java permite redirigir los flujos de entrada, salida y salida de error estándares simplemente haciendo uso de las llamadas a métodos estáticos:

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

Redirigir la salida es especialmente útil si, de repente, se desea comenzar la creación de mucha información de salida a pantalla, y el desplazamiento de la misma es demasiado rápido como para leer[3]. El redireccionamiento de la entrada es útil en programas de línea de comandos en los que se desee probar repetidamente una secuencia de entrada de usuario en particular. He aquí un ejemplo simple que muestra cómo usar estos métodos:

```
//: c11:Redireccionar.java
// Demuestra el redireccionamiento de la E/S estándar.
import java.io.*;

class Redireccionar {
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    BufferedInputStream entrada =
      new BufferedInputStream(
        new FileInputStream(
          "Redireccionar.java"));
    PrintStream salida =
      new PrintStream(
        new BufferedOutputStream(
          new FileOutputStream("prueba.out")));
    System.setIn(entrada);
    System.setOut(salida);
    System.setErr(salida);
```

[3] El Capítulo 13 muestra una solución aún más adecuada para esto: un programa de IGU con un área de desplazamiento de texto.

```
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)
      System.out.println(s);
    salida.close(); // ¡No olvidarse de esto!
  }
} ///:~
```

Este programa adjunta la entrada estándar a un archivo y redirecciona la salida estándar y la de error a otro archivo.

El redireccionamiento de la E/S manipula flujos de bytes, en vez de flujos de caracteres, por lo que se usan **InputStreams** y **OutputStreams** en vez de **Readers** y **Writers**.

Compresión

La biblioteca de E/S de Java contiene clases que dan soporte a la lectura y escritura de flujos en un formato comprimido. Estas clases están envueltas en torno a las clases de E/S existentes para proporcionar funcionalidad de compresión.

Estas clases no se derivan de las clases **Reader** y **Writer**, sino que son parte de las jerarquías **InputStream** y **OutputStream**. Esto se debe a que la biblioteca de compresión funciona con bytes en vez de caracteres. Sin embargo, uno podría verse forzado en ocasiones a mezclar ambos tipos de flujos. (Recuérdese que se puede usar **InputStreamReader** y **OutputStreamWriter** para proporcionar conversión sencilla entre un tipo y el otro.)

Clase de Compresión	Función
CheckedInputStream	**GetCheckSum( )** produce una suma de comprobación para cualquier **InputStream** (no simplemente de descompresión).
CheckedOutputStream	**GetCheckSum( )** produce una suma de comprobación para cualquier **OutputStream** (no simplemente de compresión).
DeflaterOutputStream	Clase base de las clases de compresión.
ZipOutputStream	Una **DeflaterOutputStream** que comprime datos en el formato de archivos ZIP.
GZIPOutputStream	Una **DeflaterOutputStream** que comprime datos en el formato de archivos GZIP.
InflaterInputStream	Clase base de las clases de descompresión.
ZipInflaterStream	Una **InflaterInputStream** que descomprime datos almacenados en el formato de archivos ZIP.

Clase Compresión	Función
GZIPInputStream	Una **InflaterInputStream** que descomprime datos almacenados en el formato de archivos GZIP.

Aunque hay muchos algoritmos de compresión, los más comúnmente utilizados son probablemente ZIP y GZIP. Por consiguiente, se pueden manipular de forma sencilla los datos comprimidos con las muchas herramientas disponibles para leer y escribir en estos formatos.

Compresión sencilla con GZIP

La interfaz GZIP es simple y por consiguiente la más apropiada a la hora de comprimir un único flujo de datos (en vez de tener un contenedor de datos heterogéneos). He aquí un ejemplo que comprime un archivo:

```
//: c11:ComprimirGZIP.java
// Utiliza compresion GZIP para comprimir un archivo
// cuyo nombre se pasa en línea de comandos.
import java.io.*;
import java.util.zip.*;

public class ComprimirGZIP {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    BufferedReader entrada =
      new BufferedReader(
        new FileReader(args[0]));
    BufferedOutputStream salida =
      new BufferedOutputStream(
        new GZIPOutputStream(
          new FileOutputStream("prueba.gz")));
    System.out.println("Escribiendo el archivo");
    int c;
    while((c = entrada.read()) != -1)
      salida.write(c);
    entrada.close();
    salida.close();
    System.out.println("Leyendo el archivo");
    BufferedReader entrada2 =
      new BufferedReader(
        new InputStreamReader(
          new GZIPInputStream(
            new FileInputStream("prueba.gz"))));
```

```
        String s;
        while((s = entrada2.readLine()) != null)
            System.out.println(s);
    }
} ///:~
```

El uso de clases de compresión es directo —simplemente se envuelve el flujo de salida en un **GZIPOutputStream** o en un **ZipOutputStream,** y el flujo de entrada en un **GZIPInputStream** o en un **ZipInputStream**. Todo lo demás es lectura y escritura de E/S ordinarias. Éste es un ejemplo que mezcla flujos orientados a **byte** con flujos orientados a **char**: **entrada** usa las clases **Reader**, mientras que el constructor de **GZIPOutputStream** sólo puede aceptar un objeto **OutputStream**, y no un objeto **Writer**. Cuando se abre un archivo, se convierte el **GZIPInputStream** en un **Reader**.

Almacenamiento múltiple con ZIP

La biblioteca que soporta el formato ZIP es mucho más completa. Con ella, es posible almacenar de manera sencilla múltiples archivos, e incluso existe una clase separada para hacer más sencillo el proceso de leer un archivo ZIP. La biblioteca usa el formato ZIP estándar de forma que trabaja prácticamente con todas las herramientas actualmente descargables desde Internet. El ejemplo siguiente tiene la misma forma que el anterior, pero maneja tantos parámetros de línea de comandos como se desee. Además, muestra el uso de las clases **Checksum** para calcular y verificar la suma de comprobación del archivo. Hay dos tipos de **Checksum**: **Adler32** (que es más rápido) y **CRC32** (que es más lento pero ligeramente más exacto).

```
//: c11:ComprimirZip.java
// Utiliza compresion ZIP para comprimir cualquier
// número de archivos indicados en línea de comandos.
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ComprimirZip {
    // Lanzar excepciones a la consola:
    public static void main(String[] args)
    throws IOException {
        FileOutputStream f =
            new FileOutputStream("Prueba.zip");
        CheckedOutputStream csum =
            new CheckedOutputStream(
                f, new Adler32());
        ZipOutputStream salida =
            new ZipOutputStream(
                new BufferedOutputStream(csum));
        salida.setComment("una prueba de compresión Zip con Java");
```

```
// Sin el getComment() correspondiente.
for(int i = 0; i < args.length; i++) {
  System.out.println(
    "Escribiendo el archivo " + args[i]);
  BufferedReader entrada =
    new BufferedReader(
      new FileReader(args[i]));
  salida.putNextEntry(new ZipEntry(args[i]));
  int c;
  while((c = entrada.read()) != -1)
    salida.write(c);
  entrada.close();
}
salida.close();
// ¡suma de chequeo válida únicamente una vez
// cerrado el archivo!
System.out.println("Suma de comprobacion: " +
  csum.getChecksum().getValue());
// Ahora extraer los archivos:
System.out.println("Leyendo el archivo");
FileInputStream fi =
   new FileInputStream("prueba.zip");
CheckedInputStream csumi =
  new CheckedInputStream(
    fi, new Adler32());
ZipInputStream entrada2 =
  new ZipInputStream(
    new BufferedInputStream(csumi));
ZipEntry ze;
while((ze = entrada2.getNextEntry()) != null) {
  System.out.println("Leyendo el archivo " + ze);
  int x;
  while((x = entrada2.read()) != -1)
    System.out.write(x);
}
System.out.println("Suma de comprobación: " +
  csumi.getChecksum().getValue());
entrada2.close();
// Forma alternativa de abrir y leer
// archivo zip:
ZipFile zf = new ZipFile("prueba.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
  ZipEntry ze2 = (ZipEntry)e.nextElement();
  System.out.println("Archivo: " + ze2);
```

```
        // ... extrayendo los datos como antes
      }
    }
} ///:~
```

Para cada archivo que se desee añadir al archivo, es necesario llamar a **putNextEntry( )** y pasarle un objeto **ZipEntry**. Este objeto contiene una interfaz que permite extraer y modificar todos los datos disponibles en esa entrada particular del archivo ZIP: nombre, tamaño comprimido y descomprimido, fecha, suma de comprobación CRC, datos de campos extra, comentarios, métodos de compresión y si es o no una entrada de directorio. Sin embargo, incluso aunque el formato ZIP permite poner contraseñas a los archivos, no hay soporte para esta faceta en la biblioteca ZIP de Java. Y aunque tanto **CheckedInputStream** como **CheckedOutputStream** soportan ambos sumas de comprobación, **Adler32** y **CRC32**, la clase **ZipEntry** sólo soporta una interfaz para CRC. Esto es una restricción del formato ZIP subyacente, pero podría limitar el uso de la más rápida **Adler32**.

Para extraer archivos, **ZipInputStream** tiene un método **getNextEntry( )** que devuelve la siguiente **ZipEntry** si es que la hay. Una alternativa más sucinta es la posibilidad de leer el archivo utilizando un objeto **ZipFile**, que tiene un método **entries( )** para devolver una **Enumeration** al **ZipEntries**.

Para leer la suma de comprobación hay que tener algún tipo de acceso al objeto **Checksum** asociado. Aquí, se retiene una referencia a los objetos **CheckedOutputStream** y **CheckedInputStream**, pero también se podría simplemente guardar una referencia al objeto **Checksum**.

Existe un método misterioso en los flujos Zip que es **setComment( )**. Como se mostró anteriormente, se puede poner un comentario al escribir un archivo, pero no hay forma de recuperar el comentario en el **ZipInputStream**. Parece que los comentarios están completamente soportados en una base de entrada por entrada, eso sí, sólamente vía **ZipEntry**.

Por supuesto, no hay un número de archivos al usar las bibliotecas **GZIP** o **ZIP** —se puede comprimir cualquier cosa, incluidos los datos a enviar a través de una conexión de red.

Archivos Java (JAR)

El formato ZIP también se usa en el formato de archivos JAR (Java ARchive), que es una forma de coleccionar un grupo de archivos en un único archivo comprimido, exactamente igual que el zip. Sin embargo, como todo lo demás en Java, los ficheros JAR son multiplataforma, por lo que no hay que preocuparse por aspectos de plataforma. También se pueden incluir archivos de audio e imagen, o archivo de clases.

Los archivos JAR son particularmente útiles cuando se trabaja con Internet. Antes de los archivos JAR, el navegador Web habría tenido que hacer peticiones múltiples a un servidor web para descargar todos los archivos que conforman un *applet*. Además, cada uno de estos archivos estaba sin comprimir. Combinando todos los archivos de un *applet* particular en un único archivo JAR, sólo es necesaria una petición al servidor, y la transferencia es más rápida debido a la compresión. Y cada entrada de un archivo JAR soporta firmas digitales por seguridad (consultar a la documentación de Java si se necesitan más detalles).

Un archivo JAR consta de un único archivo que contiene una colección de archivos ZIP junto con una "declaración" que los describe. (Es posible crear archivos de declaración; de otra manera el programa **jar** lo hará automáticamente.) Se puede averiguar algo más sobre declaraciones JAR en la documentación del JDK HTML.

La utilidad **jar** que viene con el JDK de Sun comprime automáticamente los archivos que se seleccionen. Se invoca en línea de comandos:

```
jar [opciones] destino [manifiesto] archivo(s)Entrada
```

Las opciones son simplemente una colección de letras (no es necesario ningún guión u otro indicador). Los usuarios de Unix/Linux notarán la semejanza con las opciones **tar**. Éstas son:

c	Crea un archivo nuevo o vacío.
t	Lista la tabla de contenidos.
x	Extrae todos los archivos.
x file	Extrae el archivo nombrado.
f	Dice: "Voy a darte el nombre del archivo." Si no lo usas, JAR asume que su entrada provendrá de la entrada estándar, o, si está creando un archivo, su salida irá a la salida estándar.
m	Dice que el primer parámetro será el nombre de un archivo de declaración creado por el usuario.
v	Genera una salida que describe qué va haciendo JAR.
O	Simplemente almacena los archivos; no los comprime (usarlo para crear un archivo JAR que se puede poner en el *classpath*).
M	No crear automáticamente un archivo de declaración.

Si se incluye algún subdirectorio en los archivos a añadir a un archivo JAR, se añade ese subdirectorio automáticamente, incluyendo también todos sus subdirectorios, etc. También se mantiene la información de rutas.

He aquí algunas formas habituales de invocar a **jar**:

```
jar cf miArchivoJar.jar *.class
```

Esto crea un fichero JAR llamado **miFicheroJar.jar** que contiene todos los archivos de clase del directorio actual, junto con un archivo declaración creado automáticamente.

```
jar cmf miArchivoJar.jar miArchivoDeclaracion.mf *.class
```

Como en el ejemplo anterior, pero añadiendo un archivo de declaración de nombre **miArchivo Declaracion.mf** creado por el usuario.

```
jar tf miArchivoJar.Jar
```

Añade el indicador que proporciona información más detallada sobre los archivos de **miArchivoJar.jar**.

```
jar cvf miAplicacion.jar audio clases imagen
```

Si se asume que **audio**, **clases** e **imagen** son subdirectorios, combina todos los subdirectorios en el archivo **miAplicacion.jar**. También se incluye el indicador que proporciona realimentación extra mientras trabaja el programa **jar**.

Si se crea un fichero JAR usando la opción **O**, el archivo se puede ubicar en el CLASSPATH:

```
CLASSPATH = "lib1.jar; lib2.jar"
```

Entonces, Java puede buscar archivos de clase en **lib1.jar** y **lib2.jar**.

La herramienta **jar** no es tan útil como una utilidad **zip**. Por ejemplo, no se pueden añadir o actualizar archivos de un archivo JAR existente; sólo se pueden crear archivos JAR de la nada. Además, no se pueden mover archivos a un archivo JAR, o borrarlos al moverlos. Sin embargo, un fichero JAR creado en una plataforma será legible transparentemente por la herramienta **jar** en cualquier otra plataforma (un problema que a veces se da en las utilidades **zip**).

Como se verá en el Capítulo 13, los archivos JAR también se utilizan para empaquetar JavaBeans.

Serialización de objetos

La *serialización de objetos* de Java permite tomar cualquier objeto que implemente la interfaz **Serializable** y convertirlo en una secuencia de bits que puede ser posteriormente restaurada para regenerar el objeto original. Esto es cierto incluso a través de una red, lo que significa que el mecanismo de serialización compensa automáticamente las diferencias entre sistemas operativos. Es decir, se puede crear un objeto en una máquina Windows, serializarlo, y enviarlo a través de la red a una máquina Unix, donde será reconstruido correctamente. No hay que preocuparse de las representaciones de los datos en las distintas máquinas, al igual que no importan la ordenación de los bytes y el resto de detalles.

Por sí misma, la serialización de objetos es interesante porque permite implementar *persitencia ligera*. Hay que recordar que la persistencia significa que el tiempo de vida de un objeto no viene determinado por el tiempo que dure la ejecución del programa —el objeto vive *mientras se den* invocaciones al mismo en el programa. Al tomar un objeto serializable y escribirlo en el disco, y luego restaurarlo cuando sea reinvocado en el programa, se puede lograr el efecto de la persistencia. La razón por la que se califica de "ligera" es porque simplemente no se puede definir un objeto utilizando algún tipo de palabra clave "persistent" y dejar que el sistema se encargue de los detalles

(aunque puede que esta posibilidad exista en el futuro). Por el contrario, hay que serializar y deserializar explícitamente los objetos.

La serialización de objetos se añadió a Java para soportar dos aspectos de mayor calibre. La invocación de Procedimientos Remotos (*Remote Method Invocation-RMI*) permite a objetos de otras máquinas comportarse como si se encontraran en la tuya propia. Al enviar mensajes a objetos remotos, es necesario serializar los parámetros y los valores de retorno. RMI se discute en el Capítulo 15.

La serialización de objetos también es necesaria en el caso de los JavaBeans, descritos en el Capítulo 13. Cuando se usa un Bean, su información de estado se suele configurar en tiempo de diseño. La información de estado debe almacenarse y recuperarse más tarde al comenzar el programa; la serialización de objetos realiza esta tarea.

La serialización de un objeto es bastante simple, siempre que el objeto implemente la interfaz **Serializable** (la interfaz es simplemente un flag y no tiene métodos). Cuando se añadió la serialización al lenguaje, se cambiaron muchas clases de la biblioteca estándar para que fueran serializables, incluidos todos los envoltorios y tipos primitivos, todas las clases contenedoras, y otras muchas. Incluso los objetos **Class** pueden ser serializados. (Véase el Capítulo 12 para comprender las implicaciones de esto.)

Para serializar un objeto, se crea algún tipo de objeto **OutputStream** y se envuelve en un **Object OutputStream**. En este momento sólo hay que invocar a **writeObject( )** y el objeto se serializa y se envía al **OutputStream**. Para invertir este proceso, se envuelve un **InputStream** en un **ObjectInputStream** y se invoca a **readObject( )**. Lo que vuelve, como siempre, es una referencia a un **Object**, así que hay que hacer una conversión hacia abajo para dejar todo como se debe.

Un aspecto particularmente inteligente de la serialización de objetos es que, no sólo salva la imagen del objeto, sino que también sigue todas las referencias contenidas en el objeto, y salva *esos* objetos, siguiendo además las referencias contenidas en cada uno de ellos, etc. A esto se le suele denominar la "telaraña de objetos" puesto que un único objeto puede estar conectado, e incluir arrays de referencias a objetos, además de objetos miembro. Si se tuviera que mantener un esquema de serialización de objetos propio, el mantenimiento del código para seguir todos estos enlaces sería casi imposible. Sin embargo, la serialización de objetos Java parece encargarse de todo haciendo uso de un algoritmo optimizado que recorre la telaraña de objetos. El ejemplo siguiente prueba el mecanismo de serialización haciendo un "gusano" de objetos enlazados, cada uno de los cuales tiene un enlace al siguiente segmento del gusano, además de un array de referencias a objetos de una clase distinta, **Datos**:

```
//: c11:Gusano.java
// Demuestra la serialización de objetos.
import java.io.*;

class Datos implements Serializable {
  private int i;
  Datos(int x) { i = x; }
  public String toString() {
    return Integer.toString(i);
```

```
    }
}

public class Gusano implements Serializable {
  // Generar un valor entero al azar:
  private static int r() {
    return (int)(Math.random() * 10);
  }
  private Datos[] d = {
    new Datos(r()), new Datos(r()), new Datos(r())
  };
  private Gusano siguiente;
  private char c;
  // Valor de i == número de segmentos
  Gusano(int i, char x) {
    System.out.println(" Constructor Gusano: " + i);
    c = x;
    if(--i > 0)
      siguiente = new Gusano(i, (char)(x + 1));
  }
  Gusano() {
    System.out.println("Constructor por defecto");
  }
  public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++)
      s += d[i].toString();
    s += ")";
    if(siguiente != null)
      s += siguiente.toString();
    return s;
  }
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws ClassNotFoundException, IOException {
    Gusano c = new Gusano(6, 'a');
    System.out.println("g = " + g);
    ObjectOutputStream salida =
      new ObjectOutputStream(
        new FileOutputStream("gusano.out"));
    salida.writeObject("Almacenamiento Gusano");
    salida.writeObject(w);
    salida.close(); // También vacía la salida
    ObjectInputStream entrada =
      new ObjectInputStream(
```

```
            new FileInputStream("gusano.out"));
        String s = (String)entrada.readObject();
        Gusano g2 = (Gusano)entrada.readObject();
        System.out.println(s + ", g2 = " + g2);
        ByteArrayOutputStream bsalida =
          new ByteArrayOutputStream();
        ObjectOutputStream salida2 =
          new ObjectOutputStream(bsalida);
        salida2.writeObject("Almacenamiento Gusano");
        salida2.writeObject(g);
        salida2.flush();
        ObjectInputStream entrada2 =
          new ObjectInputStream(
            new ByteArrayInputStream(
              bsalida.toByteArray()));
        s = (String)entrada2.readObject();
        Gusano g3 = (Gusano)entrada2.readObject();
        System.out.println(s + ", g3 = " + g3);
    }
} ///:~
```

Para hacer las cosas interesantes, el array de objetos **Datos** dentro de **Gusano** se inicializa con números al azar. (De esta forma no hay que sospechar que el compilador mantenga algún tipo de metainformación.) Cada segmento **Gusano** se etiqueta con un **char** que es automáticamente generado en el proceso de generar recursivamente la lista enlazada de **Gusanos**. Cuando se crea un **Gusano**, se indica al constructor lo largo que se desea que sea. Para hacer la referencia **siguiente** llama al constructor **Gusano** con una longitud uno menor, etc. La referencia **siguiente** final se deja a **null** indicando el final del **Gusano**.

El objetivo de todo esto era hacer algo racionalmente complejo que no pudiera ser serializado fácilmente. El acto de serializar, sin embargo, es bastante simple. Una vez que se ha creado el **ObjectOutputStream** a partir de otro flujo, **writeObject( )** serializa el objeto. Nótese que la llamada a **writeObject( )** es también para un **String**. También se pueden escribir los tipos de datos primitivos utilizando los mismos métodos que **DataOutputStream** (comparten las mismas interfaces).

Hay dos secciones de código separadas que tienen la misma apariencia. La primera lee y escribe un archivo, y la segunda escribe y lee un **ByteArray**. Se puede leer y escribir un objeto usando la serialización en cualquier **DataInputStream** o **DataOutputStream**, incluyendo, como se verá en el Capítulo 15, una red. La salida de una ejecución fue:

```
Constructor Gusano: 6
Constructor Gusano: 5
Constructor Gusano: 4
Constructor Gusano: 3
Constructor Gusano: 2
Constructor Gusano: 1
```

```
g = :a(262):b(100):c(396):d(480):e(316):f(398)
Almacenamiento Gusano, g2 =
:a(262):b(100):c(396):d(480):e(316):f(398)
Almacenamiento Gusano, g3
:a(262):b(100):c(396):d(480):e(316):f(398)
```

Se puede ver que el objeto deserializado contiene verdaderamente todos los enlaces que había en el objeto original.

Nótese que no se llama a ningún constructor, ni siquiera el constructor por defecto, en el proceso de deserialización de un objeto **Serializable**. Se restaura todo el objeto recuperando datos del **InputStream**.

La serialización de objetos está orientada al **byte**, y por consiguiente usa las jerarquías **InputStream** y **OutputStream**.

Encontrar la clase

Uno podría preguntarse qué debe tener un objeto para que pueda ser recuperado de su estado serealizado. Por ejemplo, supóngase que se serializa un objeto y se envía como un archivo o a través de una red a otro máquina. ¿Podría un programa de la otra máquina reconstruir el objeto simplemente con el contenido del archivo?

La mejor forma de contestar a esta pregunta es (como siempre) haciendo un experimento. El archivo siguiente se encuentra en el subdirectorio de este capítulo:

```
//: c11:Extraterrestre.java
// Una clase serializable.
import java.io.*;

public class Extraterrestre implements Serializable {
} ///:~
```

El archivo que crea y serializa un objeto **Extraterrestre** va en el mismo directorio:

```
//: c11:CongelarExtraterrestre.java
// Crear un archivo de salida serializado.
import java.io.*;

public class CongelarExtraterrestre {
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    ObjectOutput salida =
      new ObjectOutputStream(
        new FileOutputStream("Expediente.X"));
```

```
      Extraterrestre zorcon = new Extraterrestre();
      salida.writeObject(zorcon);
   }
} ///:~
```

Más que capturar y manejar excepciones, este programa toma el enfoque *rápido y sucio* de pasar las excepciones fuera del método **main( )**, de forma que serán reportadas en línea de comandos.

Una vez que se compila y ejecuta el programa, copie el fichero **Expediente.X** resultante al directorio denominado **expedientesx,** en el que se encuentra el siguiente código:

```
//: c11:expedientesx:DescongelarExtraterrestre.java
// Intentar recuperar un objeto serializado sin tener la
// clase de objeto almacenada en él.
import java.io.*;

public class DescongelarExtraterrestre {
   public static void main(String[] args)
   throws IOException, ClassNotFoundException {
      ObjectInputStream entrada =
         new ObjectInputStream(
            new FileInputStream("Expediente.X"));
      Object misterio = entrada.readObject();
      System.out.println(misterio.getClass());
   }
} ///:~
```

Este programa abre el archivo y lee el objeto **misterio** con éxito. Sin embargo, en cuanto se intenta averiguar algo del objeto —lo cual requiere el objeto **Class** de **Extraterrestre**— la Máquina Virtual Java (JVM) no puede encontrar **Extraterrestre.class** (a menos que esté en el Classpath, lo cual no ocurre en este ejemplo). Se obtendrá una **ClassNotFoundException.** (¡De nuevo, se desvanece toda esperanza de vida extraterrestre antes de poder encontrar una prueba de su existencia!)

Si se espera hacer mucho una vez recuperado un objeto serializado, hay que asegurarse de que la JVM pueda encontrar el archivo **.class** asociado en el path de clases locales o en cualquier otro lugar en Internet.

Controlar la serialización

Como puede verse, el mecanismo de serialización por defecto tiene un uso trivial. Pero ¿qué pasa si se tienen necesidades especiales? Quizás se tienen aspectos especiales relativos a seguridad y no se desea serializar algunas porciones de un objeto, o quizás simplemente no tiene sentido que se serialice algún subobjeto si esa parte necesita ser creada de nuevo al recuperar el objeto.

Se puede controlar el proceso de serialización implementando la interfaz **Externalizable** en vez de la interfaz **Serializable.** La interfaz **Externalizable** extiende la interfaz **Serializable** añadiendo dos métodos, **writeExternal( )** y **readExternal( ),** que son invocados automáticamente para el objeto

durante la serialización y la deserialización, de forma que se puedan llevar a cabo las operaciones especiales.

El ejemplo siguiente muestra la implementación simple de los métodos de la interfaz **Externalizable**. Nótese que **Rastro1** y **Rastro2** son casi idénticos excepto por una diferencia mínima (a ver si la descubres echando un vistazo al código):

```
//: c11:Rastros.java
// Uso simple de Externalizable & un truco.
import java.io.*;
import java.util.*;

class Rastro1 implements Externalizable {
  public Rastro1() {
    System.out.println("Constructor Rastro1");
  }
  public void writeExternal(ObjectOutput salida)
        throws IOException {
    System.out.println("Rastro1.writeExternal");
  }
  public void readExternal(ObjectInput entrada)
      throws IOException, ClassNotFoundException {
    System.out.println("Rastro1.readExternal");
  }
}

class Rastro2 implements Externalizable {
  Rastro2() {
    System.out.println("Constructor Rastro2");
  }
  public void writeExternal(ObjectOutput salida)
        throws IOException {
    System.out.println("Rastro2.writeExternal");
  }
  public void readExternal(ObjectInput entrada)
      throws IOException, ClassNotFoundException {
    System.out.println("Rastro2.readExternal");
  }
}

public class Rastros {
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Construyendo objetos:");
```

```
      Rastro1 r1 = new Rastro1();
      Rastro2 r2 = new Rastro2();
      ObjectOutputStream S =
        new ObjectOutputStream(
          new FileOutputStream("Rastros.salida"));
      System.out.println("Salvando objetos:");
      s.writeObject(r1);
      s.writeObject(r2);
      s.close();
      // Ahora recuperarlos:
      ObjectInputStream entrada =
        new ObjectInputStream(
          new FileInputStream("Rastros.salida"));
      System.out.println("Recuperando r1:");
      b1 = (Rastro1)entrada.readObject();
      // ¡OOPS! Lanza una excepción:
//! System.out.println("Recuperando r2:");
//! r2 = (Rastro2)entrada.readObject();
  }
} ///:~
```

La salida del programa es:

```
Construyendo objetos:
Constructor Rastro1
Constructor Rastro2
Salvando objetos:
Rastro1.writeExternal
Rastro2.writeExternal
Recuperando r1:
Constructor Rastro1
Rastro1.readExternal
```

La razón por la que el objeto **Rastro2** no se recupera es que intentar hacerlo causa una excepción. ¿Se ve la diferencia entre **Rastro1** y **Rastro2**? El constructor de **Rastro1** es **public**, mientras que el constructor de **Rastro2** no lo es, y eso causa la excepción en la recuperación. Puede intentarse hacer **public** el constructor de **Rastro2** y retirar los comentarios **//!** para ver los resultados correctos.

Cuando se recupera **r1**, se invoca al constructor por defecto de **Rastro1**. Esto es distinto a recuperar el objeto **Serializable**, en cuyo caso se construye el objeto completamente a partir de sus bits almacenados, sin llamadas al constructor. Con un objeto **Externalizable**, se da todo el comportamiento de construcción por defecto normal (incluyendo las inicializaciones del momento de la definición de campos), y *posteriormente*, se invoca a **readExternal( )**. Es necesario ser consciente de esto —en particular, del hecho de que siempre tiene lugar toda la construcción por defecto— para lograr el comportamiento correcto en los objetos **Externalizables**.

He aquí un ejemplo que muestra qué hay que hacer para almacenar y recuperar un objeto **Externalizable** completamente:

```
//: c11:Rastro3.java
// Reconstruyendo un objeto externalizable.
import java.io.*;
import java.util.*;

class Rastro3 implements Externalizable {
  int i;
  String s; // Sin inicialización
  public Rastro3() {
    System.out.println("Constructor Rastro3");
    // s, i sin inicializar
  }
  public Rastro3(String x, int a) {
    System.out.println("Rastro3(String x, int a)");
    s = x;
    i = a;
    // s & i inicializadas sólo en un constructor
    // distinto del constructor por defecto.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput salida)
  throws IOException {
    System.out.println("Rastro3.writeExternal");
    // Hay que hacer esto:
    salida.writeObject(s);
    salida.writeInt(i);
  }
  public void readExternal(ObjectInput entrada)
  throws IOException, ClassNotFoundException {
    System.out.println("Rastro3.readExternal");
    // Hay que hacer esto:
    s = (String)entrada.readObject();
    i = entrada.readInt();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Construyendo objetos:");
    Rastro3 r3 = new Rastro3("Una cadena ", 47);
    System.out.println(b3);
    ObjectOutputStream s =
      new ObjectOutputStream(
        new FileOutputStream("Rastro3.salida"));
```

```
        System.out.println("Salvando objeto:");
        s.writeObject(r3);
        s.close();
        // Ahora recuperarlo:
        ObjectInputStream entrada =
          new ObjectInputStream(
            new FileInputStream("Rastro3.salida"));
        System.out.println("Recuperando r3:");
        r3 = (Rastro3)entrada.readObject();
        System.out.println(r3);
    }
} ///:~
```

Los campos **s** e **i** se inicializan solamente en el segundo constructor, pero no en el constructor por defecto. Esto significa que si no se inicializan **s** e **i** en **readExternal( )**, serán **null** (puesto que el espacio de almacenamiento del objeto se inicializa a ceros en el primer paso de la creación del mismo). Si se comentan las dos líneas de código que siguen a las frases "Hay que hacer esto" y se ejecuta el programa, se verá que se recupera el objeto, **s** es **null**, e **i** es cero.

Si se está heredando de un objeto **Externalizable**, generalmente se invocará a las versiones de clase base de **writeExternal( )** y **readExternal( )** para proporcionar un almacenamiento y recuperación adecuados de los componentes de la clase base.

Por tanto, para que las cosas funcionen correctamente, no sólo hay que escribir los datos importantes del objeto durante el método **writeExternal( )** (no hay comportamiento por defecto que escriba ninguno de los objetos miembro de un objeto **Externalizable**), sino que también hay que recuperar los datos en el método **readExternal( )**. Esto puede ser un poco confuso al principio puesto que el comportamiento por defecto de la construcción de un objeto **Externalizable** puede hacer que parezca que tiene lugar automáticamente algún tipo de almacenamiento y recuperación. Esto no es así.

La palabra clave transient

Cuando se está controlando la serialización, puede ocurrir que haya un subobjeto en particular para el que no se desee que se produzca un salvado y recuperación automáticos por parte del mecanismo de serialización de Java. Éste suele ser el caso si ese objeto representa información sensible que no se desea serializar, como una contraseña. Incluso si esa información es **private** en el objeto, una vez serializada es posible que cualquiera acceda a la misma leyendo el objeto o interceptando una transmisión de red.

Una forma de evitar que partes sensibles de un objeto sean serializables es implementar la clase como **Externalizable**, como se ha visto previamente. Así, no se serializa automáticamente nada y se pueden serializar explícitamente sólo las partes de **writeExternal( )** necesarias.

Sin embargo, al trabajar con un objeto **Serializable**, toda la serialización se da automáticamente. Para controlar esto, se puede desactivar la serialización en una base campo-a-campo utilizando la palabra clave **transient**, que dice: "No te molestes en salvar o recuperar esto —me encargaré yo".

Por ejemplo, considérese un objeto **InicioSesion**, que mantiene información sobre un inicio de sesión en particular. Supóngase que, una vez verificado el inicio, se desean almacenar los datos, pero sin la contraseña. La forma más fácil de hacerlo es implementar **Serializable** y marcar el campo **contraseña** como **transient**. Debería quedar algo así:

```
//: c11:InicioSesion.java
// Demuestra la palabra clave "transient".
import java.io.*;
import java.util.*;

class InicioSesion implements Serializable {
  private Date fecha = new Date();
  private String usuario;
  private transient String contrasenia;
  InicioSesion(String nombre, String cont) {
    usuario = nombre;
    contrasenia = cont;
  }
  public String toString() {
    String cont =
      (contrasenia == null) ? "(n/a)" : contrasenia;
    return "Info inicio sesión: \n    " +
      "usuario: " + usuario +
      "\n   fecha: " + fecha +
      "\n   contrasenia: " + cont;
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    InicioSesion a = new InicioSesion("Hulk", "myLittlePony");
    System.out.println( "InicioSesion a = " + a);
      ObjectOutputStream s =
        new ObjectOutputStream(
          new FileOutputStream("InicioSesion.out"));
    s.writeObject(a);
    s.close();
    // Retraso:
    int segundos = 5;
    long t = System.currentTimeMillis()
            + seconds * 1000;
    while(System.currentTimeMillis() < t)
      ;
    // Ahora, recuperarlos:
    ObjectInputStream entrada =
      new ObjectInputStream(
        new FileInputStream("InicioSesion.out"));
```

```
        System.out.println(
          "Recuperando el objeto a las " + new Date());
        a = (InicioSesion)entrada.readObject();
        System.out.entrada( "InicioSesion a = " + a);
    }
} ///:~
```

Se puede ver que los campos **fecha** y **usuario** son normales (no **transient**) y, por consiguiente, se serializan automáticamente. Sin embargo, el campo **contraseña** es **transient**, así que no se almacena en el disco; además el mecanismo de serialización ni siquiera intenta recuperarlo. La salida es:

```
Info InicioSesion a = Info InicioSesion:
   usuario: Hulk
   fecha: Sun Mar 23 18:25:53 PST 1997
   contrasenia: myLittlePony
Recuperando objeto a las Sun Mar 23 18:25:59 PST 1997
InicioSesion a = Info InicioSesion:
   usuario: Hulk
   fecha: Sun Mar 23 18:25:53 PST 1997
   contrasenia: (n/a)
```

Cuando se recupera el objeto, el campo **contrasenia** es **null**. Nótese que **toString( )** debe comprobar si hay un valor **null** en **contrasenia** porque si se intenta ensamblar un objeto **String** haciendo uso del operador '+' sobrecargado, y ese operador encuentra una referencia a **null**, se genera una **NullPointerException**. (En versiones futuras de Java puede que se añada código que evite este problema.)

También se puede ver que el campo **fecha** se almacena y recupera a partir del disco en vez de regenerarse de nuevo.

Puesto que los objetos **Externalizable** no almacenan ninguno de sus campos por defecto, la palabra clave **transient** es para ser usada sólo con objetos **Serializable**.

Una alternativa a **Externalizable**

Si uno no es entusiasta de la implementación de la interfaz **Externalizable**, hay otro enfoque. Se puede implementar la interfaz **Serializable** y *añadir* (nótese que se dice "añadir", y no "superponer" o "implementar") métodos llamados **writeObject( )** y **readObject( ),** que serán automáticamente invocados cuando se serialice y deserialice el objeto, respectivamente. Es decir, si se proporcionan estos dos métodos, se usarán en vez de la serialización por defecto.

Estos métodos deben tener exactamente las signaturas siguientes:

```
private void
   writeObject(ObjectOutputStream flujo)
     trhrows IOException;

private void
```

```
    readObject(ObjectInputStream flujo)
      throws IOException, ClassNotFoundException
```

Desde el punto de vista del diseño, aquí todo parece un misterio. En primer lugar, se podría pensar que, debido a que estos métodos no son parte de una clase base o de la interfaz **Serializable**, deberían definirse en sus propias interface(s). Pero nótese que se definen como **private**, lo que significa que sólo van a ser invocados por miembros de esa clase. Sin embargo, de hecho no se les invoca desde otros miembros de esta clase, sino que son los métodos **writeObject( )** y **readObject ( )** de los objetos **ObjectOutputStream** y **ObjectInputStream** los que invocan a los métodos **writeObject( )** y **readObject( )** de nuestro objeto. (Nótese nuestro tremendo temor a no comenzar una larga diatriba sobre el nombre de los métodos a usar aquí. En pocas palabras: todo es confuso.) Uno podría preguntarse cómo logran los objetos **ObjectOutputStream** y **ObjectInputStream** acceso a los métodos **private** de la clase. Sólo podemos asumir que es parte de la magia de la serialización.

En cualquier caso, cualquier cosa que se defina en una **interface** es automáticamente **public,** por lo que si **writeObject( )** y **readObject( )** deben ser **private**, no pueden ser parte de una **interface**. Puesto que hay que seguir las signaturas con exactitud, el efecto es el mismo que si se está implementando una **interfaz**.

Podría parecer que cuando se invoca a **ObjectOutputStream.writeObject( )**, se interroga al objeto **Serializable** que se le pasa (utilizando sin duda la reflectividad) para ver si implementa su propio **writeObject( )**. Si es así, se salta el proceso de serialización normal, y se invoca al **writeObject( )**. En el caso de **readObject( )** ocurre exactamente igual.

Hay otra particularidad. Dentro de tu **writeObject( )** se puede elegir llevar a cabo la acción **writeObject( )** por defecto invocando a **defaultWriteObject( )**. De forma análoga, dentro de **readObject( )** se puede invocar a **defaultReadObject( )**. He aquí un ejemplo simple que demuestra cómo se puede controlar el almacenamiento y recuperación de un objeto **Serializable**:

```
//: c11:CtlSerial.java
// Controlando la serialización añadiendo métodos
// writeObject() y readObject() propios.
import java.io.*;

public class CtlSerial implements Serializable {
  String a;
  transient String b;
  public CtlSerial(String aa, String bb) {
    a = "No Transient: " + aa;
    b = "Transient: " + bb;
  }
  public String toString() {
    return a + "\n" + b;
  }
  private void
    writeObject(ObjectOutputStream flujo)
      throws IOException {
```

```
      flujo.defaultWriteObject();
      flujo.writeObject(b);
   }
   private void
      readObject(ObjectInputStream flujo)
         throws IOException, ClassNotFoundException {
      flujo.defaultReadObject();
      b = (String)flujo.readObject();
   }
   public static void main(String[] args)
   throws IOException, ClassNotFoundException {
      SerialCtl cs =
         new SerialCtl("Prueba1", "Prueba2");
      System.out.println("Antes:\n" + cs);
      ByteArrayOutputStream buf =
         new ByteArrayOutputStream();
      ObjectOutputStream s =
         new ObjectOutputStream(buf);
      s.writeObject(cs);
      // Ahora, recuperarlo:
      ObjectInputStream entrada =
         new ObjectInputStream(
            new ByteArrayInputStream(
               buf.toByteArray()));
      CtlSerial cs2 = (CtlSerial)entrada.readObject();
      System.out.println("Después:\n" + cs2);
   }
} ///:~
```

En este ejemplo, uno de los campos **String** es normal, y el otro es **transient**, para probar que se salva el campo no **transient** por parte del método **defaultWriteObject( )**, mientras que el campo **transient** se salva y recupera de forma explícita. Se inicializan los campos dentro del constructor en vez de definirlos para probar que no están siendo inicializados por ningún tipo de mecanismo automático durante la deserialización.

Si se va a usar el mecanismo por defecto para escribir las partes no **transient** del objeto, hay que invocar a **defaultWriteObject( )** como primera operación de **writeObject( )**, y a **defaultReadObject( )**, como primera operación de **readObject( )**. Éstas son llamadas extrañas a métodos. Podría parecer, por ejemplo, que está llamando al método **defaultWriteObject( )** de un **ObjectOutputStream** sin pasar argumentos, y así de algún modo convierte y conoce la referencia a su objeto y cómo escribir todas las partes no **transient**.

El almacenamiento y recuperación de objetos **transient** usa más código familiar. Y lo que es más: piense en lo que ocurre aquí. En el método **main( )**, se crea un objeto **CtlSerial**, y después se serializa a un **ObjectOutputStream**. (Nótese que en ese caso se usa un espacio de almacenamiento intermedio en vez de un archivo.) La serialización se realiza en la línea:

```
o.writeObject(cs);
```

El método **writeObject( )** debe examinar **cs** para averiguar si tiene su propio método **writeObject ( )**. (No comprobando la interfaz —pues no la hay— o el tipo de clase, sino buscando el método haciendo uso de la reflectividad.) Si lo tiene, se usa. Se sigue un enfoque semejante en el caso de **readObject( )**. Quizás ésta era la única forma, en la práctica, de solucionar el problema, pero es verdaderamente extraña.

Versionar

Es posible que se desee cambiar la versión de una clase serializable (por ejemplo, se podrían almacenar objetos de la clase original en una base de datos). Esto se soporta, pero probablemente se hará sólo en casos especiales, y requiere de una profundidad de entendimiento adicional que no trataremos de alcanzar aquí. La documentación JDK HTML descargable de *http://java.sun.com* cubre este tema de manera bastante detallada.

También se verá que muchos comentarios de la documentación JDK HTML comienzan por:

> ***Aviso**: Los objetos serializados de esta clase no serán compatibles con versiones futuras de Swing. El soporte actual para serialización es apropiado para almacenamiento a corto plazo o RMI entre aplicaciones...*

Esto se debe a que el mecanismo de versionado es demasiado simple como para que funcione correctamente en todas las situaciones, especialmente con JavaBeans. Actualmente se está trabajando en corregir su diseño, y por eso se presentan estas advertencias.

Utilizar la persistencia

Es bastante atractivo usar la tecnología de serialización para almacenar algún estado de un programa de forma que se pueda restaurar el programa al estado actual más adelante. Pero antes de poder hacer esto hay que resolver varias cuestiones. ¿Qué ocurre si se serializan dos objetos teniendo ambos una referencia a un tercero? Cuando se restauren esos dos objetos de su estado serializado ¿se obtiene sólo una ocurrencia del tercer objeto? ¿Qué ocurre si se serializan los dos objetos en archivos separados y se deserializan en partes distintas del código?

He aquí un ejemplo que muestra el problema:

```
//: c11:MiMundo.java
import java.io.*;
import java.util.*;

class Casa implements Serializable {}

class Animal implements Serializable {
  String nombre;
  Casa casaFavorita;
  Animal(String nm, Casa h) {
```

```
      nombre = nm;
      casaFavorita = h;
   }
   public String toString() {
      return nombre + "[" + super.toString() +
         "], " + casaFavorita + "\n";
   }
}

public class MiMundo {
   public static void main(String[] args)
   throws IOException, ClassNotFoundException {
      Casa casa = new Casa();
      ArrayList  animales = new ArrayList();
      animales.add(
         new Animal("Bosco el perro", casa));
      animales.add(
         new Animal("Ralph el hamster", casa));
      animales.add(
         new Animal("Fronk el gato", casa));
      System.out.println("animales: " + animales);

      ByteArrayOutputStream buf1 =
         new ByteArrayOutputStream();
      ObjectOutputStream s1 =
         new ObjectOutputStream(buf1);
      s1.writeObject(animales);
      s1.writeObject(animales); // Escribir un 2° conjunto
      // Escribir a un flujo distinto:
      ByteArrayOutputStream buf2 =
         new ByteArrayOutputStream();
      ObjectOutputStream s2 =
         new ObjectOutputStream(buf2);
      s2.writeObject(animales);
      // Ahora, recuperarlos:
      ObjectInputStream entrada1 =
         new ObjectInputStream(
            new ByteArrayInputStream(
               buf1.toByteArray()));
      ObjectInputStream entrada2 =
         new ObjectInputStream(
            new ByteArrayInputStream(
               buf2.toByteArray()));
      ArrayList animales1 =
         (ArrayList)entrada1.readObject();
```

```
      ArrayList animales2 =
         (ArrayList)entrada1.readObject();
      ArrayList animales3 =
         (ArrayList)entrada2.readObject();
      System.out.println("animales1: " + animales1);
      System.out.println("animales2: " + animales2);
      System.out.println("animales3: " + animales3);
   }
} ///:~
```

Una de las cosas interesantes aquí es que es posible usar la serialización de objetos para y desde un array de bytes logrando una "copia en profundidad" de cualquier objeto **Serializable**. (Una copia en profundidad implica duplicar la telaraña de objetos entera, en vez de simplemente el objeto básico y sus referencias). La copia se cubre en detalle en el Apéndice A.

Los objetos **Animal** contienen campos del tipo **Casa**. En el método **main( )**, se crea un **ArrayList** de estos **Animales** y se serializa dos veces en un flujo y de nuevo a otro flujo distinto. Cuando se deserializan e imprimen, se verán en la ejecución los resultados siguientes (en cada ejecución, los objetos estarán en distintas posiciones de memoria):

```
animales: [Bosco el perro[Animal@1cc76c], Casa@1cc769
, Ralph el hamster[Animal@1cc76d], Casa@1cc769
, Fronk el gato[Animal@1cc76e], Casa@1cc769
]
animales1: [Bosco el perro[Animal@1cca0c], Casa@1cca16
, Ralph el hamster[Animal@1cca17], Casa@1cca16
, Fronk el gato[Animal@1cca1b], Casa@1cca16
]
animales2: [Bosco el perro[Animal@1cca0c], Casa@1cca16
, Ralph el hamster[Animal@1cca17], Casa@1cca16
, Fronk el gato[Animal@1cca1b], Casa@1cca16
]
animales3: [Bosco el perro[Animal@1cca52], Casa@1cca5c
, Ralph el hamster[Animal@1cca5d], Casa@1cca5c
, Fronk el gato[Animal@1cca61], Casa@1cca5c
]
```

Por supuesto, se puede esperar que los objetos deserializados tengan direcciones distintas a la del original. Pero nótese que en **animales1** y **animales2** aparecen las mismas direcciones, incluyendo las referencias al objeto **Casa** que ambos comparten. Por otro lado, cuando se recupera **animales3** el sistema no puede saber que los objetos del otro flujo son alias de los objetos del primer flujo, por lo que construye una telaraña de objetos completamente diferente.

Mientras se serialice todo a un único flujo, se podrá recuperar la misma telaraña de objetos que se escribió, sin duplicaciones accidentales de los mismos. Por supuesto, se puede cambiar el estado de los objetos en el tiempo que transcurre entre la escritura del primero y el último, pero eso es res-

ponsabilidad de cada uno —los objetos se escribirán en el estado en el que estén (y con cualquier conexión que tengan con otros objetos) en el preciso momento de la serialización.

Lo más seguro si se desea salvar el estado de un sistema es hacer la serialización como una operación "atómica". Si se serializa una parte, se hacen otras cosas, luego se serializa otra parte, etc., no se estará almacenando el sistema de forma segura. Lo que hay que hacer es poner todos los objetos que conforman el estado del sistema en un único contenedor y simplemente se escribe este contenedor en una única operación. Después, es posible restaurarlo también con una única llamada a un método.

El ejemplo siguiente es un sistema de diseño asistido por ordenador (CAD) que demuestra el enfoque. Además, se introduce en el aspecto de los campos **static** —si se echa un vistazo a la documentación se verá que **Class** es **Serializable**, por lo que debería ser fácil almacenar campos **static** simplemente serializando el objeto **Class**. De cualquier forma, este enfoque parece sensato.

```
//: c11:EstadoCAD.java
// Almacenando y restaurando el estado de un
// sistema CAD aparente.
import java.io.*;
import java.util.*;

abstract class Figura implements Serializable {
  public static final int
    ROJO = 1, AZUL = 2, VERDE = 3;
  private int xPos, yPos, dimension;
  private static Random r = new Random();
  private static int contador = 0;
  abstract public void setColor(int nuevoColor);
  abstract public int getColor();
  public Figura(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  public String toString() {
    return getClass() +
      " color[" + getColor() +
      "] xPos[" + xPos +
      "] yPos[" + yPos +
      "] dim[" + dimension + "]\n";
  }
  public static Figura factoriaAleatoria() {
    int xVal = r.nextInt() % 100;
    int yVal = r.nextInt() % 100;
    int dim = r.nextInt() % 100;
    switch(contador++ % 3) {
```

```
        default:
        case 0: return new Circulo(xVal, yVal, dim);
        case 1: return new Cuadrado(xVal, yVal, dim);
        case 2: return new Linea(xVal, yVal, dim);
      }
    }
}

class Circulo extends Figura {
  private static int color = ROJO;
  public Circulo(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void estableceColor(int nuevoColor) {
    color = nuevoColor;
  }
  public int obtenerColor() {
    return color;
  }
}

class Cuadrado extends Figura {
  private static int color;
  public Cuadrado(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    color = ROJO;
  }
  public void establecerColor(int nuevoColor) {
    color = nuevoColor;
  }
  public int obtenerColor() {
    return color;
  }
}

class Linea extends Figura {
  private static int color = ROJO;
  public static void
  serializarEstadoEstatico(ObjectOutputStream os)
      throws IOException {
    os.writeInt(color);
  }
  public static void
  deserializarEstadoEstatico(ObjectInputStream os)
      throws IOException {
```

```
      color = os.readInt();
   }
   public Linea(int xVal, int yVal, int dim) {
      super(xVal, yVal, dim);
   }
   public void establecerColor(int nuevoColor) {
      color = nuevoColor;
   }
   public int obtenerColor() {
      return color;
   }
}

public class EstadoCAD {
   public static void main(String[] args)
   throws Exception {
      ArrayList tiposFigura, figuras;
      if(args.length == 0) {
         tiposFigura = new ArrayList();
         figura = new ArrayList();
         // Añadir referencias a objetos de la clase:
         tiposFigura.add(Circulo.class);
         tiposFigura.add(Cuadrado.class);
         tiposFigura.add(Linea.class);
         // Construir algunas figuras:
         for(int i = 0; i < 10; i++)
            figuras.add(Figura.factoriaAleatoria());
         // Poner todos los colores estáticos a VERDE:
         for(int i = 0; i < 10; i++)
            ((Figura)figuras.get(i))
               .establecerColor(Poligono.VERDE);
         // Salvar el vector de estado:
         ObjectOutputStream salida =
            new ObjectOutputStream(
               new FileOutputStream("EstadoCAD.out"));
         salida.writeObject(Tiposfigura);
         Linea.serializacionEstadoEstatico(salida);
         salida.writeObject(figuras);
      } else { // Hay un parámetro de línea de comandos
         ObjectInputStream entrada =
            new ObjectInputStream(
               new FileInputStream(args[0]));
         // Leer de la misma forma en que se escribieron:
         tiposFigura = (ArrayList)entrada.readObject();
         Linea.deserializarEstadoEstatico(entrada);
```

```
        Figuras = (ArrayList)entrada.readObject();
      }
      // Mostrar los polígonos:
      System.out.println(figuras);
    }
} ///:~
```

La clase **Figura implementa Serializable**, por lo que cualquier cosa que herede de **Figura** es automáticamente también **Serializable**. Cada **Figura** contiene datos, y cada clase **Figura** derivada contiene un campo **static** que determina el color de todos esos tipos de **Figuras**. (Colocar un campo **static** en la clase base resultaría en un solo campo, mientras que los campos **static** no se duplican en las clases derivadas). Se pueden superponer los métodos de la clase base para establecer el color de los diversos tipos (los métodos **static** no se asignan dinámicamente, por lo que son método normales). El método **factoriaAleatoria( )** crea una **Figura** diferente cada vez que se le invoca, utilizando valores al azar para los datos **Figura**.

Círculo y **Cuadrado** son extensiones directas de **Figura**; la única diferencia radica en que **Círculo** inicializa **color** en el momento de su definición y **Cuadrado** lo inicializa en el constructor. Se dejará la discusión sobre **Línea** para un poco más adelante.

En el método **main( )**, se usa un **ArrayList** para guardar los objetos **Class** y otro para mantener las figuras. Si no se proporciona un parámetro de línea de comandos, se crea el **ArrayList tiposFigura** y se añaden los objetos **Class**, para posteriormente crear el **ArrayList figuras** y añadirle objetos **Figura**. A continuación, se ponen a **VERDE** todos los valores de **static color**, y todo se serializa al archivo **EstadoCAD.out**.

Si se proporciona un parámetro de línea de comandos (se supone que **EstadoCAD.out**) se abre ese archivo y se usa para restaurar el estado del programa. En ambas situaciones, se imprime el **ArrayList** de **Figuras**. Los resultados de una ejecución son:

```
<java EstadoCAD
[class Circulo color[3] xPos[-51] yPos[-99] dim[38]
, class Cuadrado color[3] xPos[2] yPos[61] dim[-46]
, class Linea color[3] xPos[51] yPos[73] dim[64]
, class Circulo color[3] xPos[-70] yPos][1] dim[16]
, class Cuadrado color[3] xPos[3] yPos[94] dim[-36]
, class Linea color[3] xPos[-80] yPos[-21] dim[-35]
, class Circulo color[3] xPos[-75] yPos[-43] dim[22]
, class Cuadrado color[3] xPos[81] yPos[30] dim[-45]
, class Linea color[3] xPos[-29] yPox[92] dim[17]
, class Circulo color[3] xPos[17] yPox[90] dim[-76]
]

<java EstadoCAD EstadoCAD.out
[class Circulo color[1] xPos[-51] yPos[-99] dim[38]
, class Cuadrado color[0] xPos[2] yPos[61] dim[-46]
, class Linea color[3] xPos[51] yPos[73] dim[64]
```

```
, class Circulo color[1] xPos[-70] yPos][1] dim[16]
, class Cuadrado color[0] xPos[3] yPos[94] dim[-36]
, class Linea color[3] xPos[-80] yPos[-21] dim[-35]
, class Circulo color[1] xPos[-75] yPos[-43] dim[22]
, class Cuadrado color[0] xPos[81] yPos[30] dim[-45]
, class Linea color[3] xPos[-29] yPox[92] dim[17]
, class Circulo color[1] xPos[17] yPox[90] dim[-76]
]
```

Se puede ver que los valores de **xPos**, **yPos** y **dim** se almacenaron y recuperaron satisfactoriamente, pero hay algo que no va bien al recuperar la información **static**. Se están introduciendo todo "3", pero no vuelve a visualizarse así. Los **Círculos** tienen valor 1 (**ROJO**, que es la definición), y los **Cuadrados** tienen valor 0 (recuérdese que se inicializan en el constructor). ¡Es como si, de hecho, los **statics** no se serializaran! Esto es así —incluso aunque la clase **Class** sea **Serializable**, no hace lo que se espera. Por tanto si se desea serializar **statics**, hay que hacerlo a mano.

Esto es para lo que sirven los métodos **static serializarEstadoEstatico( )** y **deserializar EstadoEstatico( )** de **Línea**. Se puede ver que se invocan explícitamente como parte del proceso de almacenamiento y recuperación. (Nótese que el orden de escritura al archivo serializado y el de lectura del mismo debe ser igual.) Por consiguiente, para que **EstadoCAD.java** funcione correctamente hay que:

1. Añadir un **serializarEstadoEstatico( )** y un **deserializarEstadoEstatico( )** a los polígonos.
2. Eliminar el **ArrayList tiposFigura** y todo el código relacionado con él.
3. Añadir llamadas a los nuevos métodos estáticos para serializar y deserializar en cada figura.

Otro aspecto a tener en cuenta es la seguridad, puesto que la serialización también almacena datos **private**. Si se tiene un problema de seguridad, habría que marcar los campos afectados como **transient**. Pero entonces hay que diseñar una forma segura de almacenar esa información, de forma que cuando se restaure, se puedan poner a cero esas variables **private**.

Identificar símbolos de una entrada

IdentificarSimbolos es el proceso de dividir una secuencia de caracteres en una secuencia de "símbolos", que son bits de texto delimitados por lo que se elija. Por ejemplo, los símbolos podrían ser palabras, pudiendo delimitarse por un espacio en blanco y signos de puntuación. Las dos clases proporcionadas por la biblioteca estándar de Java y que pueden ser usadas para poner símbolos son: **StreamTokenizer** y **StringTokenizer**.

StreamTokenizer

Aunque **StreamTokenizer** no deriva de **InputStream** ni de **OutputStream**, sólo funciona con objetos **InputStream**, por lo que pertenece a la parte de E/S de la biblioteca.

Considérese un programa que cuenta la ocurrencia de cada palabra en un archivo de texto:

```
//: c11:RecuentoPalabra.java
// Cuenta las palabras de un archivo, muestra los
// resultados de forma ordenada.
import java.io.*;
import java.util.*;

class Contador {
  private int i = 1;
  int leer() { return i; }
  void incrementar() { i++; }
}

public class RecuentoPalabra {
  private FileReader archivo;
  private StreamTokenizer st;
  // Un TreeMap mantiene las claves ordenadas:
  private TreeMap cuentas = new TreeMap();
  RecuentoPalabra(String nombreArchivo)
    throws FileNotFoundException {
    try {
      archivo = new FileReader(nombreArchivo);
      st = new StreamTokenizer(
        new BufferedReader(archivo));
      st.ordinaryChar('.');
      st.ordinaryChar('-');
    } catch(FileNotFoundException e) {
      System.err.println(
        "No se pudo abrir " + nombreArchivo);
      throw e;
    }
  }
  void limpieza() {
    try {
      archivo.close();
    } catch(IOException e) {
      System.err.println(
        "archivo.close() sin exito");
    }
  }
  void contarPalabras() {
    try {
      while(st.nextToken() !=
        StreamTokenizer.TT_EOF) {
        String s;
        switch(st.ttype) {
```

```
            case StreamTokenizer.TT_EOL:
              s = new String("EOL");
              break;
            case StreamTokenizer.TT_NUMBER:
              s = Double.toString(st.nval);
              break;
            case StreamTokenizer.TT_WORD:
              s = st.sval; // Es ya un String
              break;
            default: // sólo hay un carácter en ttype
              s = String.valueOf((char)st.ttype);
          }
          if(cuentas.containsKey(s))
            ((Contador)cuentas.get(s)).incrementar();
          else
            cuentas.put(s, new Contador());
        }
      } catch(IOException e) {
        System.err.println(
          "st.nextToken() sin exito");
      }
    }
    Collection valores() {
      return cuentas.values();
    }
    Set conjuntoClaves() { return cuentas.keySet(); }
    Contador obtenerContador(String s) {
      return (Contador)cuentas.get(s);
    }
    public static void main(String[] args)
    throws FileNotFoundException {
      RecuentoPalabra rp =
        new RecuentoPalabra(args[0]);
      rp.contarPalabras();
      Iterator claves = rp.conjuntoClaves().iterator();
      while(claves.hasNext()) {
        String clave = (String)claves.next();
        System.out.println(clave + ": "
                 + cp.obtenerContador(clave).read());
      }
      rp.limpieza();
    }
} ///:~
```

Es fácil presentar las palabras de forma ordenada almacenando los datos en un **TreeMap**, que organiza automáticamente sus claves en orden (véase Capítulo 9). Cuando se logra un conjunto de claves utilizando **keySet( )**, éstas también estarán en orden.

Para abrir el archivo, se usa un **FileReader**, y para convertir el archivo en palabras se crea un **StreamTokenizer** a partir del **FileReader** envuelto en un **BufferedReader**. En **Stream Tokenizer**, hay una lista de separadores por defecto, y se pueden añadir más con un conjunto de métodos. Aquí, se usa **ordinaryChar( )** para decir: "Este carácter no tiene el significado en el que estoy interesado", por lo que el analizador no lo incluye como parte de las palabras que crea. Por ejemplo, decir **st.ordinaryChar('.')** quiere decir que no se incluirán los periodos como partes de las palabras a analizar. Se puede encontrar más información en la documentación JDK HTML de *http://java.sun.com*.

En **contarPalabras( )**, se sacan los símbolos de uno en uno desde el flujo, y se usa la información **ttype** para determinar qué hacer con cada símbolo, puesto que un símbolo puede ser un fin de línea, un número, una cadena de caracteres, o un único carácter.

Una vez que se encuentra un símbolo, se pregunta al **TreeMap cuentas** para ver si ya contiene el símbolo como clave. Si lo tiene, se incrementa el objeto **Contador** correspondiente, para indicar que se ha encontrado otra instancia de esa palabra. Si no, se crea un nuevo **Contador** —puesto que el constructor de **Contador** inicializa su valor a uno, esto también sirve para contar la palabra.

RecuentoPalabra no es un tipo de **TreeMap**, por lo que no heredó de éste. Lleva a cabo un tipo de funcionalidad especifico del tipo, por lo que incluso aunque hay que reexponer los métodos **keys( )** y **values( )**, eso sigue sin querer decir que debería usarse la herencia, puesto que utilizar varios métodos de **TreeMap** sería inadecuado. Además, se usan otros métodos como **obtenerContador( )**, que obtiene el **Contador** de un **String** en particular, y **sortedKeys( )**, que produce un **Iterator**, para finalizar el cambio en la forma de la interfaz de **RecuentoPalabra**.

En el método **main( )** se puede ver el uso de un **RecuentoPalabra** para abrir y contar las palabras de un archivo —simplemente ocupa dos líneas de código. Después se extrae un **Iterator** a una lista de claves (palabras) ordenadas, y se usa éste para extraer otra clave y su **Contador** asociado. La llamada a **limpieza( )** es necesaria para asegurar que se cierre el archivo.

StringTokenizer

Aunque éste no forma parte de la biblioteca de E/S, el **StringTokenizer** tiene funcionalidad lo suficientemente similar a **StreamTokenizer** como para describirlo aquí.

El **StringTokenizer** devuelve todos los símbolos contenidos en una cadena de caracteres de uno en uno. Estos símbolos son caracteres consecutivos delimitados por tabuladores, espacios y saltos de línea. Por consiguiente, los símbolos de la cadena "¿Dónde está mi gato?" son "¿Dónde", "está", "mi", y "gato?". Al igual que con **StreamTokenizer** se puede indicar a **StringTokenizer** que divida la entrada de la forma que desee, pero con **StringTokenizer** esto se logra pasando un segundo parámetro al constructor, que es un **String** con los delimitadores que se desea utilizar. En general, si se necesita más sofisticación, hay que usar un **StreamTokenizer**.

Para pedir a un **StringTokenizer** que te pase el siguiente token de la cadena se usa el método **nextToken( )** que o bien devuelve el símbolo, o bien una cadena de caracteres vacía para indicar que no quedan más símbolos.

A modo de ejemplo, el programa siguiente lleva a cabo un análisis limitado de una sentencia, buscando secuencias de frases clave para indicar si hay algún tipo de alegría o tristeza implicadas.

```
//: c11:AnalizarSentencia.java
// Buscar secuencias particulares en sentencias.
import java.util.*;

public class AnalizarSentencia {
  public static void main(String[] args) {
    analizar("Yo estoy contento por esto");
    analizar("Yo no estoy contento por esto");
    analizar("¡No lo estoy! Estoy contento");
    analizar("Yo no estoy triste por esto");
    analizar("Yo estoy triste por esto");
    analizar("¡No lo estoy! Estoy triste");
    analizar("¿Estas tu contento por esto?");
    analizar("¿Estas tu triste por esto?");
    analizar("¡Eres tu! Estoy contento");
    analizar("¡Eres tu! Estoy triste");
  }
  static StringTokenizer st;
  static void analizar(String s) {
    prt("\nnueva frase >> " + s);
    boolean triste = false;
    st = new StringTokenizer(s);
    while (st.hasMoreTokens()) {
      String símbolo = siguiente();
      // Buscar hasta encontrar uno de los dos
      // símbolos de inicio:
      if(!simbolo.equals("Yo") &&
         !simbolo.equals("¡Estas"))
        continue; // Parte de arriba del bucle while
      if(símbolo.equals("Yo")) {
        String s2 = siguiente();
        if(!s2.equals("estoy")) // Debe estar después de "yo"
          break; // Salir del bucle while
        else {
          String s3 = siguiente();
          if(s3.equals("triste")) {
            triste = true;
            break; // Salir del bucle while
```

```
            }
            if (s3.equals("no")) {
              String s4 = siguiente();
              if(s4.equals("triste"))
                break; // Dejar triste a false
              if(s4.equals("contento")) {
                triste = true;
                break;
              }
            }
          }
        }
        if(símbolo.equals("Estás")) {
          String s2 = siguiente();
          if(!s2.equals("tu"))
            break; // Debe ir después de éstas
          String s3 = siguiente();
          if(s3.equals("triste"))
            triste = true;
          break; // Salir del bucle while
        }
      }
      if(triste) prt("Tristeza detectada");
    }
    static String siguiente() {
      if(st.hasMoreTokens()) {
        String s = st.nextToken();
        prt(s);
        return s;
      }
      else
        return "";
    }
    static void prt(String s) {
      System.out.println(s);
    }
} ///:~
```

Por cada cadena de caracteres que se analiza, se entra en un bucle **while** y se extraen de la cadena los símbolos. Nótese la primera sentencia **if**, que dice que se **continúe** (volver al principio del bucle y comenzar de nuevo) si el símbolo no es ni "Yo" ni "Estás". Esto significa que cogerá símbolos hasta que se encuentre un "Yo" o un "Estás". Se podría pensar en usar == en vez del método **equals( )**, pero eso no funcionaría correctamente, pues == compara los valores de las referencias, mientras que **equals( )** compara contenidos.

La lógica del resto del método **analizar( )** es que el patrón que se está buscando es "Yo estoy triste", "Yo no estoy contento" o "¿Tú estás triste? Sin la sentencia **break**, el código habría sido aún más complicado de lo que ya es. Habría que ser conscientes de que un analizador típico (éste es un ejemplo primitivo de uno) normalmente tiene una tabla de estos símbolos y un fragmento de código que se mueve a través de los estados de la tabla a medida que se leen los nuevos símbolos.

Debería pensarse que **StringTokenizer** sólo es un atajo para un tipo simple y específico de **StreamTokenizer**. Sin embargo, si se tiene un **String** en el que se desean identificar símbolos, y **StringTokenizer** es demasiado limitado, todo lo que hay que hacer es convertirlo en un stream con **StringBufferInputStream** y después usarlo para crear **StreamTokenizer** mucho más potente.

Comprobar el estilo de escritura de mayúsculas

En esta sección, echaremos un vistazo a un ejemplo más completo del uso de la E/S de Java, que también hace uso de la identificación de símbolos. Este proyecto es directamente útil pues lleva a cabo una comprobación de estilo para asegurarse que el uso de mayúsculas se adecua al estilo de Java, tal y como se relata en *http://java.sun.com/docs/codeconv/index.html.* Abre cada archivo **.java** del directorio actual y extrae todos los nombres de archivos e identificadores, para mostrar después las que no utilicen el estilo Java.

Para que el programa funcione correctamente, hay que construir, en primer lugar, un repositorio de nombres de clases para guardar todos los nombres de clases de la biblioteca estándar de Java. Esto se logra moviendo todos los subdirectorios de código fuente de la biblioteca estándar de Java y ejecutando **ExploradorClases** en cada subdirectorio. Deben proporcionarse como argumentos el nombre del repositorio (usando la misma trayectoria y nombre siempre) y la opción de línea de comandos **–a** para indicar que deberían añadirse los nombres de clases al repositorio.

Al usar el programa para que compruebe el código de cada uno, hay que pasarle la trayectoria y el nombre del repositorio que debe usar. Comprobará todas las clases e identificadores en el directorio actual y dirá cuáles no siguen el ejemplo de uso de mayúsculas típico de Java.

Uno debería ser consciente de que el programa no es perfecto; pocas veces señalará algo que piensa que es un problema, pero que mirando al código se comprobará que no hay que cambiar nada. Esto es un poco confuso, pero sigue siendo más sencillo que intentar encontrar todos estos casos simplemente mirando cuidadosamente al código.

```
//: c11:ExploradorClases.java
// Busca clases e identificadores en todos los archivos
// de un directorio para comprobar el uso de mayúsculas.
// Asume listados de código que compilan adecuadamente.
// No lo hace todo bien pero es una ayuda
// útil.
import java.io.*;
import java.util.*;

class Mapa MultiCadena extends HashMap {
```

```
   public void add(String key, String value) {
      if(!containsKey(key))
         put(key, new ArrayList());
      ((ArrayList)get(key)).add(value);
   }
   public ArrayList getArrayList(String key) {
      if(!containsKey(key)) {
         System.err.println(
            "ERROR: no se puede encontrar la clave: " + key);
         System.exit(1);
      }
      return (ArrayList)get(key);
   }
   public void printValues(PrintStream p) {
      Iterator k = keySet().iterator();
      while(k.hasNext()) {
         String unaClave = (String)k.next();
         ArrayList val = getArrayList(oneKey);
         for(int i = 0; i < val.size(); i++)
            p.println((String)val.get(i));
      }
   }
}

public class ExploradorClases {
   private File ruta;
   private String[] listaArchivos;
   private Properties clases = new Properties();
   private MapaMultiCadena
      mapaClases = new MapaMultiCadena(),
      mapaIdent = new MapaMultiCadena();
   private StreamTokenizer entrada;
   public ExploradorClases() throws IOException {
      ruta = new File(".");
      listaArchivos = ruta.list(new FiltroJava());
      for(int i = 0; i < listaArchivos.length; i++) {
         System.out.println(listaArchivos[i]);
         try {
            explorarListado(listaArchivos[i]);
         } catch(FileNotFoundException e) {
            System.err.println("No se pudo abrir " +
               listaArchivos[i]);
         }
      }
   }
```

```
  void explorarListado(String nombreF)
  throws IOException {
    entrada = new StreamTokenizer(
        new BufferedReader(
          new FileReader(nombreF)));
    // Parece que no funciona:
    // entrada.slashStarComments(true);
    // entrada.slashSlashComments(true);
    entrada.ordinaryChar('/');
    entrada.ordinaryChar('.');
    entrada.wordChars('_', '_');
    entrada.eolIsSignificant(true);
    while(entrada.nextToken() !=
          StreamTokenizer.TT_EOF) {
      if(entrada.ttype == '/')
        comerComentarios();
      else if(entrada.ttype ==
              StreamTokenizer.TT_WORD) {
        if(entrada.sval.equals("class") ||
           entrada.sval.equals("interface")) {
          // Conseguir el nombre de la clase:
             while(entrada.nextToken() !=
                   StreamTokenizer.TT_EOF
                   && entrada.ttype !=
                   StreamTokenizer.TT_WORD)
               ;
             clases.put(entrada.sval, entrada.sval);
             mapaClases.add(nombreF, entrada.sval);
        }
        if(entrada.sval.equals("import") ||
           entrada.sval.equals("package"))
          descartarLinea();
        else // Es un identificador o palabra clave
          mapaIdent.add(nombreF, entrada.sval);
      }
    }
  }
  void descartarLinea() throws IOException {
    while(entrada.nextToken() !=
          StreamTokenizer.TT_EOF
          && entrada.ttype !=
          StreamTokenizer.TT_EOL)
      ; // Lanza tokens al final de la línea
  }
  // La retirada del comentario de StreamTokenizer
```

```
  // parecia rota. Esto los extrae:
  void comerComentarios() throws IOException {
    if(entrada.nextToken() !=
        StreamTokenizer.TT_EOF) {
      if(entrada.ttype == '/')
        descartarLinea();
      else if(entrada.ttype != '*')
        entrada.pushBack();
      else
        while(true) {
          if(entrada.nextToken() ==
            StreamTokenizer.TT_EOF)
            break;
          if(entrada.ttype == '*')
            if(entrada.nextToken() !=
              StreamTokenizer.TT_EOF
              && entrada.ttype == '/')
              break;
        }
    }
  }
  public String[] nombresClases() {
    String[] resultado = new String[clases.size()];
    Iterator e = clases.keySet().iterator();
    int i = 0;
    while(e.hasNext())
      resultado[i++] = (String)e.next();
    return result;
  }
  public void comprobarNombresClases() {
    Iterator archivos = mapaClases.keySet().iterator();
    while(archivos.hasNext()) {
      String archivo = (String)archivos.next();
      ArrayList cls = mapaClases.getArrayList(archivo);
      for(int i = 0; i < cls.size(); i++) {
        String nombreClase = (String)cls.get(i);
        if(Character.isLowerCase(
             nombreClase.charAt(0)))
          System.out.println(
            "error de escritura de mayusculas, archivo: "
            + archivo + ", clase: "
            + nombreClase);
      }
    }
  }
```

```
public void comprobarNombreIdent() {
  Iterator archivos = mapaIdent.keySet().iterator();
  ArrayList conjuntoInformes = new ArrayList();
  while(archivos.hasNext()) {
    String archivo = (String)archivos.next();
    ArrayList ids = mapaIdent.getArrayList(archivo);
    for(int i = 0; i < ids.size(); i++) {
      String id = (String)ids.get(i);
      if(!clases.contains(id)) {
        // Ignorar identificadores de longitud 3 o
        // mayor que sean todo mayúsculas
        // (probablemente son valores static final):
        if(id.length() >= 3 &&
           id.equals(
             id.toUpperCase()))
          continue;
        // Comprobar para ver si el primer carácter es mayúscula:
        if(Character.isUpperCase(id.charAt(0))){
          if(conjuntoInformes.indexOf(archivo + id)
              == -1){ // No informado aun
            conjuntoInformes.add(archivo + id);
            System.out.println(
              "Error de mayúscula indentada en:"
              + archivo + ", ident: " + id);
          }
        }
      }
    }
  }
}
static final String uso =
  "Uso: \n" +
  "ExploradorClases nombresClases -a\n" +
  "\tañade todos los nombres de clase de este \n" +
  "\tdirectorio al archivo repositorio \n" +
  "\tllamado 'nombresClases'\n" +
  "ExploradorClases nombresClases\n" +
  "\tComprueba todos los archivos java de este \n" +
  "\tdirectorio buscando errores de escritura de mayúsculas, \n" +
  "\tusando el archivo repositorio 'nombresClases'";
private static void uso() {
  System.err.println(uso);
  System.exit(1);
}
public static void main(String[] args)
```

```
    throws IOException {
      if(args.length < 1 || args.length > 2)
        uso();
      ExploradorClases c = new ExploradorClases();
      File antiguo = new File(args[0]);
      if(antiguo.exists()) {
        try {
          // Intentar abrir un archivo de
          // propiedades existente:
          InputStream antiguolistado =
            new BufferedInputStream(
              new FileInputStream(antiguo));
          c.clases.load(antiguoListado);
          antiguoListado.close();
        } catch(IOException e) {
          System.err.println("No se pudo abrir "
            + antiguo + " en modo lectura");
          System.exit(1);
        }
      }
      if(args.length == 1) {
        c.comprobarNombresClases();
        c.comprobarNombresClases();
      }
      // Escribir los nombres de clase en un repositorio:
      if(args.length == 2) {
        if(!args[1].equals("-a"))
          uso();
        try {
          BufferedOutputStream salida =
            new BufferedOutputStream(
              new FileOutputStream(args[0]));
          c.clases.store(salida,
            "ExploradorClases.java encontro clases");
          salida.close();
        } catch(IOException e) {
          System.err.println(
            "No se pudo escribir " + args[0]);
          System.exit(1);
        }
      }
    }
  }

  class FiltroJava implements FilenameFilter {
```

```
  public boolean aceptar(File dir, String nombre) {
    // Eliminar información de trayectoria:
    String f = new File(nombre).getName();
    return f.trim().endsWith(".java");
  }
} ///:~
```

La clase **MapaMultiCadena** es una herramienta que permite establecer una correspondencia entre un grupo de cadenas de caracteres y su clave. Usa un **HashMap** (esta vez con herencia) con la clave como única cadena de caracteres con correspondencias sobre **ArrayList**. El método **add( )** simplemente comprueba si ya hay una clave en el **HashMap**, y si no, pone una. El método **getArrayList( )** produce un **ArrayList** para una clave en particular, y **printValues( )**, que es especialmente útil para depuración, imprime todos los valores de **ArrayList** en **ArrayList**.

Para que todo sea sencillo, se ponen todos los nombres de la biblioteca estándar de Java en un objeto **Properties** (de la biblioteca estándar de Java). Recuérdese que un objeto **Properties** es un **HashMap** que sólo guarda objetos **String,** tanto para las entradas de clave como para la de valor. Sin embargo, se puede salvar y restaurar a disco con una única llamada a un método, por lo que es ideal para un repositorio de nombres. De hecho, sólo necesitamos una lista de nombres, y un **HashMap** no puede aceptar **null,** ni para su entrada clave, ni para su entrada valor. Por tanto, se usará el mismo objeto tanto para la clave como para valor.

Para las clases e identificadores que se descubran para los archivos en un directorio en particular, se usan dos **MultiStringMaps**: **mapaClases** y **mapaIdent**. Además, cuando el programa empieza carga el repositorio de nombres de clase estándares en el objeto **Properties** llamado **clases**, y cuando se encuentra un nuevo nombre de clase en el directorio local se añade tanto a **clases** como a **mapaClases**. De esta forma, puede usarse **mapaClases** para recorrer todas las clases en el directorio local, y puede usarse **clases** para ver si el símbolo actual es un nombre de clase (que indica que comienza una definición de un objeto o un método).

El constructor por defecto de **ExploradorClases** crea una lista de nombres de archivo usando la implementación **FiltroJava** de **FilenameFilter**, mostrada al final del archivo. Después llama a **explorarListado( )** para cada nombre de archivo.

Dentro de **explorarListado( )** se abre el código fuente y se convierte en **StreamTokenizer**. En la documentación, se supone que pasar **true** a **slashStarComments( )** y **slashSlashComments( )** retira estos comentarios, pero parece un poco fraudulento, pues no funciona muy bien. En vez de ello, las líneas se marcan como comentarios que son extraídos por otro método. Para hacer esto, el "/" debe capturarse como un carácter normal, en vez de dejar a **StreamTokenizer** que lo absorba como parte de un comentario, y el método **ordinaryChar( )** dice al **StreamTokenizer** que lo haga así. Esto también funciona para los puntos ("."), puesto que se desea retirar las llamadas a métodos en forma de identificadores individuales. Sin embargo, el guión bajo, que suele ser tratado por **StreamTokenizer** como un carácter individual, debería dejarse como parte de los identificadores, pues aparece en valores **static final**, como **TT_EOF**, etc., usados en este mismo programa. El método **wordChars( )** toma un rango de caracteres que se desee añadir a los ya dejados dentro del símbolo a *analizar* como palabras. Finalmente, al analizar comentarios de una línea o descartar una lí-

nea, hay que saber cuándo se produce un fin de línea[4], por lo que se llama a **eolIsSignificant(true)** que mostrará los finales de línea en vez de dejar que sean absorbidos por el **StreamTokenizer**.

El resto de **explorarListado( )** lee y vuelve a actuar sobre los símbolos hasta el fin de fichero, que se encuentra cuando **nextToken( )** devuelva el valor **final static StreamTokenizer.TT_EOF**.

Si el símbolo es un "/" es potencialmente un comentario, por lo que se llama a **comerComentarios( )** para que lo maneje. Únicamente la otra situación que nos interesa en este caso es si es una palabra, donde pueden presentarse varios casos.

Si la palabra es **class** o **interfaz**, el siguiente símbolo representa un nombre de clase o interfaz, y se introduce en **clases** y **mapaClases**. Si la palabra es **import** o **package**, entonces no se desea el resto de la línea. Cualquier otra cosa debe ser un identificador (que nos interesa) o una palabra clave (que no nos interesa, pero que en cualquier caso se escriben con minúsculas, por lo que no pasa nada por incluirlas). Éstas se añaden a **mapaIdent**.

El método **descartarLínea( )** es una simple herramienta que busca finales de línea. Nótese que cada vez que se encuentre un nuevo símbolo, hay que comprobar los finales de línea.

El método **comerComentarios( )** es invocado siempre que se encuentra un "/" en el bucle de análisis principal. Sin embargo, eso no quiere decir necesariamente que se haya encontrado un comentario, por lo que hay que extraer el siguiente comentario para ver si hay otra barra diagonal (en cuyo caso se descarta toda la línea) o un asterisco. Si no estamos ante ninguna de éstas, ¡hay que volver a insertar el símbolo que se acaba de extraer! Afortunadamente, el método **pushBack( )** permite volver a introducir el símbolo actual en el flujo de entrada, de forma que cuando el bucle de análisis principal llame a **nextToken( )**, se obtendrá el que se acaba de introducir.

Por conveniencia, el método **nombresClases( )** produce un array de todos los nombres del contenedor **clases**. Este método no se usa en el programa pero es útil en procesos de depuración.

Los dos siguiente métodos son precisamente aquéllos en los que de hecho se realiza la comprobación . En **comprobarNombresClases( )**, se extraen los nombres de clase de **mapaClases** (que, recuérdese, contiene sólo los nombres de este directorio, organizados por nombre de archivo, de forma que se puede imprimir el nombre de archivo junto con el nombre de clase errante). Esto se logra extrayendo cada **ArrayList** asociado y recorriéndolo, tratando de ver si el primer carácter está en minúsculas. Si es así, se imprimirá el pertinente mensaje de error.

En **comprobarNombresIdent( )**, se sigue un enfoque similar: se extrae cada nombre de identificador de **mapaIdent**. Si el nombre no está en la lista **clases**, se asume que es un identificador o una palabra clave. Se comprueba un caso especial: si la longitud del identificador es tres o más *y* todos sus caracteres son mayúsculas, se ignora el identificador pues es probablemente un valor **static final** como **TT_EOF**. Por supuesto, éste no es un algoritmo perfecto, pero asume que generalmente los identificadores formados exclusivamente por letras mayúsculas se pueden ignorar.

---

[4] N. del traductor: en inglés *End Of Line* o *EOL*.

En vez de informar de todos los identificadores que empiecen con una mayúscula, este método mantiene un seguimiento de aquéllos para los que ya se ha generado un informe en un **ArrayList** denominado **conjuntoInformes( )**. Éste trata al **ArrayList** como un "conjunto" que indica si un elemento se encuentra o no en el conjunto. El elemento se genera concatenando el nombre de archivo y el identificador. Si el elemento no está en el conjunto, se añade y después se emite el informe.

El resto del listado se lleva a cabo en el método **main( )**, que se mantiene ocupado manejando los parámetros de línea de comandos y averiguando si se está o no construyendo un repositorio de nombres de clase a partir de la biblioteca estándar de Java o comprobando la validez del código escrito. En ambos casos hace un objeto **ExploradorClase**.

Se esté construyendo un repositorio o utilizando uno, hay que intentar abrir el repositorio existente. Haciendo un objeto **File** y comprobando su existencia, se puede decidir si abrir un archivo y **load( )** las **clases** de la lista de **Properties** dentro de **ExploradorClase**. (Las clases del repositorio se añaden, más que sobreescribirse, a las clases encontradas en el constructor **ExploradorClase**.) Si se proporciona sólo un parámetro en línea de comandos, se quiere llevar a cabo una comprobación de nombres de clase e identificadores, pero si se proporcionan dos argumentos (siendo el segundo "**-a**") se está construyendo un repositorio de nombres de clase. En este caso, se abre un archivo de salida y se usa el método **Properties.save( )** para escribir la lista en un archivo, junto con una cadena de caracteres que proporciona información de cabecera de archivo.

Resumen

La biblioteca de flujos de E/S de Java satisface los requisitos básicos: se puede llevar a cabo lectura y escritura con la consola, un archivo, un bloque de memoria o incluso a través de Internet (como se verá en el Capítulo 15). Con la herencia, se pueden crear nuevos tipos de objetos de entrada y salida. E incluso se puede añadir una extensibilidad simple a los tipos de objetos que aceptará un flujo redefiniendo el método **toString( )** que se invoca automáticamente cuando se pasa un objeto a un método que esté esperando un **String** (la "conversión automática de tipos" limitada de Java).

En la documentación y diseño de la biblioteca de flujos de E/S quedan cuestiones sin contestar. Por ejemplo, habría sido genial si se pudiese decir que se desea que se lance una excepción si se intenta sobreescribir un archivo cuando se abre como salida —algunos sistemas de programación permiten especificar que se desea abrir un archivo de salida, pero sólo si no existe aún. En Java, parece suponerse que uno usará un objeto **File** para determinar si existe un archivo, porque si se abre como un **FileOutputStream** o **FileWriter** siempre será sobreescrito.

La biblioteca de flujos de E/S trae a la mente sentimientos entremezclados; hace gran parte del trabajo y es portable. Pero si no se entiende ya el patrón decorador, el diseño no es intuitivo, por lo que hay una gran sobrecarga en lo que a aprendizaje y enseñanza de la misma se refiere. También está incompleta: no hay soporte para dar formato a la salida, soportado casi por el resto de paquetes de E/S del resto de lenguajes.

Sin embargo, una vez que *se entiende* el patrón decorador, y se empieza a usar la biblioteca en situaciones que requieren su flexibilidad, se puede empezar a beneficiar de este diseño, punto en el que su coste en líneas extra puede no molestar tanto.

Si no se encuentra lo que se estaba buscando en este capítulo (que no ha sido más que una introducción, que no pretendía ser comprensivo) se puede encontrar información en profundidad en *Java I/O*, de Elliote Rusty Harold (O'Reilly, 1999).

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Abrir un archivo de texto de forma que se pueda leer del mismo de línea en línea. Leer cada línea como un **String** y ubicar ese objeto **String** en un **LinkedList**. Imprimir todas las líneas del **LinkedList** en orden inverso.

2. Modificar el Ejercicio 1 de forma que el nombre del archivo que se lea sea proporcionado como un parámetro de línea de comandos.

3. Modificar el Ejercicio 2 para que también abra un archivo de texto de forma que se pueda escribir texto en el mismo. Escribir las líneas del **ArrayList**, junto con los números de línea (no intentar usar las clases "LineNumber"), fuera del archivo.

4. Modificar el Ejercicio 2 para forzar que todas las líneas de **ArrayList** estén en mayúsculas y enviar los resultados a **System.out**.

5. Modificar el Ejercicio 2 para que tome palabras a buscar dentro del archivo como parámetros adicionales de línea de comandos. Imprimir todas las líneas en las que casen las palabras.

6. Modificar **ListadoDirectorio.java** de forma que **FilenameFilter** abra cada archivo y acepte el archivo basado en la existencia de alguno de los parámetros de la línea de comandos en ese archivo.

7. Crear una clase denominada **ListadoDirectorioOrdenado** con un constructor que tome información de una ruta de archivo y construya un listado de directorio ordenado con todos los archivos de esa ruta. Crear dos métodos **listar( )** sobrecargados que, bien produzcan toda la lista, o bien un subconjunto de la misma basándose en un argumento. Añadir un método **tamanio( )** que tome un nombre de archivo y produzca el tamaño de ese archivo.

8. Modificar **RecuentoPalabra.java** para que produzca un orden alfabético en su lugar, utilizando la herramienta del Capítulo 9.

9. Modificar **RecuentoPalabra.java** de forma que use una clase que contenga un **String** y un valor de cuenta para almacenar cada palabra, y un **Set** de esos objetos para mantener la lista de palabras.

10. Modificar **DemoFlujoES.java** de forma que use **LineNumberInputStream** para hacer un seguimiento del recuento de líneas. Nótese que es mucho más fácil mantener el seguimiento desde la programación.

11. Basándonos en la Sección 4 de **DemoFlujoES.java,** escribir un programa que compare el rendimiento de escribir en un archivo al usar E/S con y sin espacios de almacenamiento intermedio.

12. Modificar la Sección 5 de **DemoFlujoES.java** para eliminar los espacios en la línea producida por la primera llamada a **entrada5br.readLine( )**. Hacerlo utilizando un bucle **while** y **readChar( )**.

13. Reparar el programa **EstadoCAD.java** tal y como se describe en el texto.

14. En **Rastros.java**, copiar el archivo y renombrarlo a **ComprobarRastro.java**, y renombrar la clase **Rastro2** a **ComprobarRastro** (haciéndola además **public** y retirando el ámbito público de la clase **Rastros** en el proceso). Eliminar las marcas **//!** del final del archivo y ejecutar el programa incluyendo las líneas que causaban ofensa. A continuación, marcar como comentario el constructor por defecto de **ComprobarRastro**. Ejecutarlo y explicar por qué funciona. Nótese que tras compilar, hay que ejecutar el programa con "**java Rastros**" porque el método **main( )** sigue en la clase **Rastros.**

15. En **Rastro3.java**, marcar como comentarios las dos líneas tras las frases "Hay que hacer esto:" y ejecutar el programa. Explicar el resultado y por qué difiere de cuando las dos líneas se encuentran en el programa.

16. (Intermedio) En el Capítulo 8, localizar el ejemplo **ControlesCasaVerde.java**, que consiste en tres archivos. En **ControlesCasaVerde.java**, la clase interna **Rearrancar( )** tiene un conjunto de eventos duramente codificados. Cambiar el programa de forma que lea los eventos y sus horas relativas desde un archivo de texto. (Desafío: utilizar un *método factoría* de patrones de diseño para construir los eventos —véase *Thinking in Patterns with Java*, descargable desde *http://www.BruceEckel.com.*)

12: Identificación de tipos en tiempo de ejecución

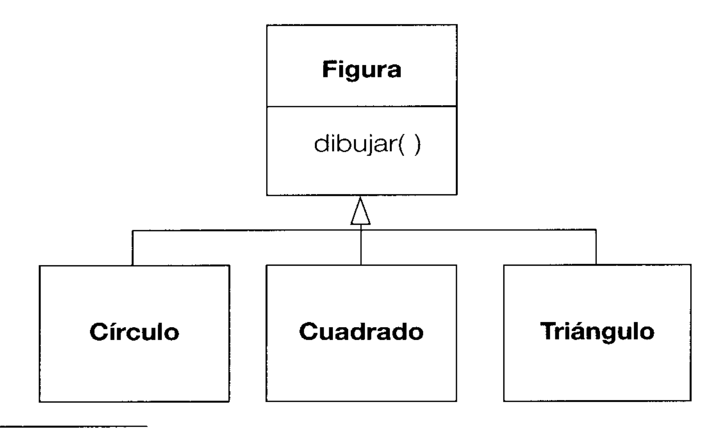
La idea de identificación de tipos en tiempo de ejecución[1] parece bastante simple a primera vista: permite encontrar el tipo exacto de un objeto cuando se tiene sólo una referencia al tipo base.

Sin embargo, la *necesidad* de RTTI no cubre una gran plétora de aspectos de diseño OO interesantes (y en ocasiones que dejan perplejos), y destapa preguntas fundamentales sobre cómo se deberían estructurar los programas.

Este capítulo repasa las formas en que Java permite descubrir información, tanto sobre clases, como relativas a objetos en tiempo de ejecución. Esto se hace de dos formas: RTTI "tradicional", que asume que todos los tipos están disponibles tanto en tiempo de ejecución como de compilación, y el mecanismo de "reflexión", que permite descubrir información de clases únicamente en tiempo de ejecución. Se cubrirá primero el RTTI "tradicional", siguiendo una discusión sobre la reflectividad a continuación.

La necesidad de RTTI

Considérese el ya familiar ejemplo de una jerarquía de clases que hace uso del polimorfismo. El tipo genérico es el de la clase base **Figura**, y los tipos específicos derivados son **Círculo**, **Cuadrado**, y **Triángulo**:



[1] N. del traductor: En inglés *Run-time Type Identification* o RTTI.

Éste es un diagrama jerárquico de clases típico, con la clase base en la parte superior y las clases derivadas creciendo hacia abajo. La meta normal en la POO es que la mayor cantidad posible de código manipule referencias de la clase base (en este caso, **Figura**) de forma que si se decide extender el programa añadiendo una clase nueva (**Romboide**, derivada de **Figura**, por ejemplo), la gran mayoría del código no se vea afectada. En este ejemplo, el método asignado estáticamente en la interfaz **Figura** es **dibujar( ),** por tanto, se pretende que el programador cliente invoque a **dibujar( )** a través de una referencia **Figura** genérica. El método **dibujar( )** está superpuesto en todas las clases derivadas, y dado que por ello es un método de correspondencia dinámica, se obtendrá el comportamiento adecuado incluso aunque se invoque a través de una referencia **Figura** genérica. Esto es el polimorfismo.

Por consiguiente, se suele crear un objeto específico (**Círculo**, **Cuadrado** o **Triángulo**), se aplica un molde hacia arriba a una **Figura** (olvidando el tipo específico del objeto), y se usa como una referencia **Figura** anónima en el resto del programa.

Para dar un breve repaso al polimorfismo y aplicar un molde hacia arriba, se puede echar un vistazo al siguiente ejemplo:

```
//: c12:Figuras.java
import java.util.*;

class Figura {
  void dibujar() {
    System.out.println(this + ".dibujar()");
  }
}

class Circulo extends Figura {
  public String toString() { return "Circulo"; }
}

class Cuadrado extends Figura {
  public String toString() { return "Cuadrado"; }
}

class Triangulo extends Figura{
  public String toString() { return "Triangulo"; }
}

public class Figura {
  public static void main(String[] args) {
    ArrayList s = new ArrayList();
    s.add(new Circulo());
    s.add(new Cuadrado());
    s.add(new Triangulo());
```

```
      Iterator e = s.iterator();
      while(e.hasNext())
         ((Figura)e.next()).dibujar();
   }
} ///:~
```

La clase base contiene un método **dibujar( )** que usa indirectamente **toString( )** para imprimir un identificador de la clase pasando **this** a **System.out.println( )**. Si esa función ve un objeto, llama automáticamente al método **toString( )** para producir una representacion **String.**

Cada una de las clases derivadas superpone el método **toString( )** (de **Object**) de forma que **dibujar( )** acaba imprimiendo algo distinto en cada caso. En el método **main( )** se crean tipos específicos de **Figura** que después se añaden a una **ArrayList**. Éste es el momento en el que se aplica un molde hacia arriba puesto que **ArrayList** sólo guarda **Objects**. Puesto que en Java todo es un **Object** (excepto los tipos primitivos), un **ArrayList** puede guardar también objetos **Figura**. Pero al aplicar un molde hacia arriba a **Object**, también se pierde información específica, incluyendo el hecho de que los objetos son **Figuras**. En lo que a **ArrayList** se refiere, son simplemente **Objects**.

En el momento de recuperar un elemento de **ArrayList** con **next( )**, todo se vuelve más complicado. Dado que **ArrayList** simplemente guarda **Objects**, naturalmente **next( )** produce una referencia **Object**. Pero sabemos que verdaderamente es una referencia a **Figura**, y se desea poder enviar a ese objeto **Figura** mensajes. Por tanto, es necesaria una conversión a **Figura** utilizando el molde tradicional **"(Figura)"**. Ésta es la forma más básica de RTTI, puesto que en tiempo de ejecución se comprueba que todas las conversiones sean correctas. Esto es exactamente lo que significa RTTI: identificar en tiempo de ejecución el tipo de los objetos.

En este caso, la conversión RTTI es sólo parcial: se convierte el **Object** a **Figura**, y no hasta **Círculo**, **Cuadrado** o **Triángulo**. Esto se debe a que lo único que *se sabe* en este momento es que **ArrayList** está lleno de **Figuras**.

En tiempo de compilación, se refuerza esto sólo por reglas autoimpuestas; en tiempo de ejecución, prácticamente se asegura.

Ahora toma su papel el polimorfismo y se determina el método exacto invocado para **Figura** para saber si es una referencia a **Círculo**, **Cuadrado** o **Triángulo**. Y así es como debería ser en general; se desea que la mayor parte del código sepa lo menos posible sobre los tipos *específicos* de los objetos, y simplemente tratar con la representación general de una familia de objetos (en este caso, **Figura**). El resultado es un código más fácil de escribir, leer y mantener, y los diseños serán más fáciles de implementar, entender y cambiar. Por tanto, el polimorfismo es la meta general en la programación orientada a objetos.

Pero, ¿qué ocurre si se tiene un problema de programación especial que es más fácil de solucionar conociendo el tipo exacto de una referencia genérica? Por ejemplo, supóngase que se desea permitir a los objetos resaltar todas las formas de determinado tipo pintándolas de violeta. De esta forma se podría, por ejemplo, localizar todos los triángulos de la pantalla. Esto es lo que logra RTTI: se puede pedir a una referencia **Figura** el tipo exacto al que se refiere.

El objeto **Class**

Para entender cómo funciona la RTTI en Java, hay que saber primero cómo se representa la información de tipos en tiempo de ejecución. Esto se logra mediante una clase especial de objeto denominada el *objeto Class*, que contiene información sobre la clase. (En ocasiones se le denomina *metaclase*.) De hecho, se usa el objeto **Class** para crear todos los objetos "normales" de la clase.

Hay un objeto **Class** por cada clase que forme parte del programa. Es decir, cada vez que se escribe y compila una nueva clase, se crea también un objeto **Class** (y se almacena, en un archivo de nombre igual y extensión **.class**). En tiempo de ejecución, cuando se desea construir un objeto de esa clase, la Máquina Virtual Java que está ejecutando el programa comprueba en primer lugar si se ha cargado el objeto **Class** para ese tipo. Sino, la JVM lo carga localizando el archivo **.class** de este nombre. Por consiguiente, los programas Java no se cargan completamente antes de empezar, a diferencia de la mayoría de lenguajes tradicionales.

Una vez que el objeto **Class** de ese tipo está en memoria, se usa para crear todos los objetos de ese tipo.

Si esto parece un poco sombrío o uno no puede creerlo, he aquí un programa demostración que lo prueba:

```
//: c12:Confiteria.java
// Examen de cómo funciona el cargador de clases.

class Caramelo {
   static {
      System.out.println("Cargando Caramelo");
   }
}

class Chicle {
   static {
      System.out.println("Cargando Chicle");
   }
}

class Galleta {
   static {
      System.out.println("Cargando Galleta");
   }
}

public class Confiteria {
   public static void main(String[] args) {
      System.out.println("dentro del método main");
      new Caramelo();
```

```
      System.out.println("Después de crear Caramelo");
      try {
        Class.forName("Chicle");
      } catch(ClassNotFoundException e) {
        e.printStackTrace(System.err);
      }
      System.out.println(
        "Después de Class.forName(\"Chicle\")");
      new Galleta();
      System.out.println("Después de crear la Galleta");
  }
} ///:~
```

Cada una de las clases **Caramelo**, **Chicle** y **Galleta** tienen una cláusula **static** que se ejecuta al cargar la clase por primera vez. Se imprimirá información para indicar cuándo se carga esa clase. En el método **main( )**, las creaciones de objetos están dispersas entre sentencias de impresión para ayudar a detectar el momento de carga.

Una línea particularmente interesante es:

```
Class.forName("Chicle");
```

Este método es un miembro **static** de **Class** (al que pertenecen todos los objetos **Class**). Un objeto **Class** es como cualquier otro objeto, de forma que se pueden recuperar y manipular referencias al mismo. (Eso es lo que hace el cargador.) Una de las formas de lograr una referencia al objeto **Class** es **forName( )**, que toma un **String** que contiene el nombre textual (¡hay que tener cuidado con el deletreo y las mayúsculas!) de la clase particular para la que se desea una referencia. Devuelve una referencia **Class.**

La salida de este programa para una JVM es:

```
dentro del método main
Cargando Caramelo
Despues de crear Caramelo
Cargando Chicle
Despues de Class.forName("Chicle")
Cargando Galleta
Despues de crear Galleta
```

Se puede ver que cada objeto **Class** sólo se crea cuando es necesario, y se lleva a cabo la inicialización **static** en el momento de cargar la clase.

Literales de clase

Java proporciona una segunda forma de producir la referencia al objeto **Class**, utilizando un *literal de clase*. En el programa de arriba, esto sería de la forma:

```
Chicle.class;
```

que no sólo es más simple, sino que es también seguro puesto que se comprueba en tiempo de compilación. Dado que elimina la llamada al método, también es más eficiente.

Los literales de clase funcionan con clases regulares además de con interfaces, arrays y tipos primitivos. Además, hay un campo estándar denominado **TYPE** que existe para cada una de las clases envoltorio primitivas. El campo **TYPE** produce una referencia al objeto **Class** para el tipo primitivo asociado, como:

... es equivalente a ...	
boolean.class	**Boolean.TYPE**
char.class	**Character.TYPE**
byte.class	**Byte.TYPE**
short.class	**Short.TYPE**
int.class	**Integer.TYPE**
long.class	**Long.TYPE**
float.class	**Float.TYPE**
double.class	**Double.TYPE**
void.class	**Void.TYPE**

Es preferible usar las versiones "**.class**" si se puede, puesto que son más consistentes con clases regulares.

Comprobar antes de una conversión

Hasta la fecha, se han visto las formas RTTI incluyendo:

1. La conversión clásica; por ejemplo, "**(Figura)**," que usa RTTI para asegurarse de que la conversión sea correcta y lanza una **ClassCastException** si se ha hecho una conversión errónea.

2. El objeto **Class** que representa el tipo de objeto. Se puede preguntar al objeto **Class** por información útil de tiempo de ejecución.

En C++, la conversión clásica **"(Figura)"** *no* lleva a cabo RTTI. Simplemente dice al compilador que trate al objeto como el nuevo tipo. En Java, que lleva a cabo la comprobación de tipos, a esta comprobación se le suele llamar una "conversión segura hacia abajo".

La razón por la que se usa la expresión "*aplicar molde hacia abajo*" es la disposición histórica del diagrama de jerarquía de clases. Si convertir un **Círculo** a una **Figura** es aplicar un molde hacia arriba, convertir una **Figura** en un **Círculo** es aplicar un molde hacia abajo. Sin embargo, se sabe que un **Círculo** es también una **Figura,** y el compilador permite libremente una asignación hacia

arriba, pero se *desconoce* que una **Figura** sea necesariamente un **Círculo**, de forma que el compilador no te permite llevar a cabo una asignación hacia abajo sin usar una conversión explícita.

Hay una tercera forma de RTTI en Java. Es la palabra clave **instanceof** la que dice si un objeto es una instancia de un tipo particular. Devuelve un **boolean,** de forma que si se usa en forma de cuestión, como en:

```
if(x instanceof Perro)
  ((Perro) x).ladrar();
```

la sentencia **if** de arriba comprueba si el objeto **x** pertenece a la clase **Perro** *antes* de convertir **x** en un **Perro**. Es importante utilizar **instanceof** antes de aplicar un molde hacia abajo, cuando no se tiene ninguna otra información que indique el tipo de objeto; de otra forma se acabará con una **ClassCastException**.

De forma ordinaria, se podría estar buscando un tipo (triángulos para pintarlos de violeta, por ejemplo), pero se puede llevar fácilmente la cuenta de *todos* los objetos utilizando **instanceof**. Supóngase que se tiene una familia de clases **AnimalDomestico**:

```
//: c12:AnimalesDomesticos.java
class AnimalDomestico {}
class Perro extends AnimalDomestico {}
class Doguillo extends Perro {}
class Gato extends AnimalDomestico {}
class Roedor extends AnimalDomestico {}
class Gerbo extends Roedor {}
class Hamster extends Roedor {}

class Contador { int i; } ///:~
```

La clase **Contador** se usa para llevar un seguimiento de cualquier tipo de **AnimalDomestico** particular. Se podría pensar que es como un **Integer** que puede ser modificado.

Utilizando **instanceof** pueden contarse todos los animales domésticos:

```
//: c12:RecuentoAnimalDomestico.java
// Usando instanceof.
import java.util.*;

public class RecuentoAnimalDomestico {
  static String[] nombresTipo = {
    "AnimalDomestico", "Perro", "Doguillo", "Gato",
    "Roedor", "Gerbo", "Hamster",
  };
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws Exception {
```

```
ArrayList animalesDomesticos = new ArrayList();
try {
  Class[] tiposAnimalDomestico = {
    Class.forName("Perro"),
    Class.forName("Doguillo"),
    Class.forName("Gato"),
    Class.forName("Roedor"),
    Class.forName("Gerbil"),
    Class.forName("Hamster"),
  };
  for(int i = 0; i < 15; i++)
    animalesDomesticos.add(
      tiposAnimalDomestico[
        (int)(Math.random()*tiposAnimalDomestico.length)]
        .newInstance());
} catch(InstantiationException e) {
  System.err.println("No se puede instanciar");
  throw e;
} catch(IllegalAccessException e) {
  System.err.println("No se puede acceder");
  throw e;
} catch(ClassNotFoundException e) {
  System.err.println("No se puede encontrar la clase");
  throw e;
}
HashMap h = new HashMap();
for(int i = 0; i < nombresTipo.length; i++)
  h.put(nombresTipo[i], new Contador());
for(int i = 0; i < animalesDomesticos.size(); i++) {
  Object o = animalesDomesticos.get(i);
  if(o instanceof AnimalDomestico)
    ((Contador)h.get("animalesDomesticos")).i++;
  if(o instanceof Perro)
    ((Contador)h.get("Perro")).i++;
  if(o instanceof Doguillo)
    ((Contador)h.get("Doguillo")).i++;
  if(o instanceof Gato)
    ((Contador)h.get("Gato")).i++;
  if(o instanceof Roedor)
    ((Contador)h.get("Roedor")).i++;
  if(o instanceof Gerbo)
    ((Contador)h.get("Gerbo")).i++;
  if(o instanceof Hamster)
    ((Contador)h.get("Hamster")).i++;
}
```

```
      for(int i = 0; i < animalesDomesticos.size(); i++)
        System.out.println(animalesDomesticos.get(i).getClass());
      for(int i = 0; i < nombresTipo.length; i++)
        System.out.println(
          nombresTipo[i] + " cantidad: " +
          ((Contador)h.get(nombresTipo[i])).i);
  }
} ///:~
```

Hay una restricción bastante severa en **instanceof**: se puede comparar sólo a tipos con nombre, y no a un objeto **Class**. En el ejemplo de arriba se podría pensar que es tedioso escribir todas esas expresiones **instanceof**, lo que es cierto. Pero no hay forma de automatizar inteligentemente **instanceof** creando una **ArrayList** de objetos **Class** y compararlo con éstos en su lugar (permanezca atento —hay una alternativa). Ésta no es una restricción tan grande como se podría pensar, porque generalmente se entenderá que el diseño será más defectuoso si se acaban escribiendo una multitud de expresiones **instanceof**.

Por supuesto, este ejemplo es artificial —probablemente se pondría un miembro de datos **static** en cada tipo de incremento en el constructor para mantener un seguimiento del recuento. Se haría algo así *si* se tuviera control sobre el código fuente de la clase y se cambiaría. Dado que éste no es siempre el caso, RTTI puede venir bien.

Utilizar literales de clase

Es interesante ver cómo se puede reescribir el ejemplo **RecuentoAnimalDomestico.java** utilizando literales de clase. El resultado es más limpio en muchos sentidos:

```
//: c12:RecuentoAnimalDomestico2.java
// Utilizando literales de clase.
import java.util.*;

public class RecuentoAnimalDomestico2 {
  public static void main(String[] args)
  throws Exception {
    ArrayList animalesDomesticos = new ArrayList();
    Class[] tiposAnimalDomestico = {
      // Literales de clase:
      AnimalDomestico.class,
      Perro.class,
      Doguillo.class,
      Gato.class,
      Roedor.class,
      Gerbo.class,
      Hamster.class,
    };
    try {
```

```
      for(int i = 0; i < 15; i++) {
        // Desplazamiento de uno para eliminar AnimalDomestico.class:
        int rnd = 1 + (int)(
          Math.random() * (tiposAnimalDomestico.length - 1));
        pets.add(
          tiposAnimalDomestico[rnd].newInstance());
      }
    } catch(InstantiationException e) {
      System.err.println("No se puede instanciar");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("No se puede acceder");
      throw e;
    }
    HashMap h = new HashMap();
    for(int i = 0; i < tiposAnimalDomestico.length; i++)
      h.put(tiposAnimalDomestico[i].toString(),
        new Contador());
    for(int i = 0; i < animalesDomesticos.size(); i++) {
      Object o = animalesDomesticos.get(i);
      if(o instanceof animalDomestico)
        ((Contador)h.get("clase AnimalDomestico")).i++;
      if(o instanceof Perro)
        ((Contador)h.get("class Perro")).i++;
      if(o instanceof Doguillo)
        ((Contador)h.get("class Doguillo")).i++;
      if(o instanceof Gato)
        ((Contador)h.get("class Gato")).i++;
      if(o instanceof Roedor)
        ((Contador)h.get("class Roedor")).i++;
      if(o instanceof Gerbo)
        ((Contador)h.get("class Gerbo")).i++;
      if(o instanceof Hamster)
        ((Contador)h.get("class Hamster")).i++;
    }
    for(int i = 0; i < animalesDomesticos.size(); i++)
      System.out.println(animalesDomesticos.get(i).getClass());
    Iterator claves = h.keySet().iterator();
    while(claves.hasNext()) {
      String nm = (String)claves.next();
      Contador cnt = (Contador)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " cantidad: " + cnt.i);
    }
```

```
  }
} ///:~
```

Aquí, se ha retirado el array **nombresTipo** para conseguir los strings de nombres de tipos de los objetos **Class**. Nótese que el sistema puede distinguir entre clases e interfaces.

También se puede ver que la creación de **tiposAnimalDomestico** no tiene por qué estar rodeada de un bloque **try** porque se evalúa en tiempo de compilación y por consiguiente no lanzará ninguna excepción, a diferencia de **Class.forName( )**.

Cuando se crean dinámicamente los objetos **AnimalDomestico**, se puede ver que se restringe el número aleatorio de forma que esté entre uno y **tiposAnimalDomestico.length** y sin incluir el cero. Eso es porque el cero hace referencia a **AnimalDomestico.class**, y presumiblemente un objeto **AnimalDomestico** genérico no sea interesante. Sin embargo, dado que **AnimalDomestico.class** es parte de **tiposAnimalesDomestico** el resultado es que se cuentan todos los animales domésticos.

Un **instanceof** dinámico

El método de **Class isInstance** proporciona una forma de invocar dinámicamente al operador **instanceof**. Por consiguiente, todas esas tediosas sentencias **instanceof** pueden eliminarse en el ejemplo **RecuentoAnimalDomestico**:

```
//: c12:RecuentoAnimalDomestico3.java
// Usando isInstance().
import java.util.*;

public class RecuentoAnimalDomestico3 {
  public static void main(String[] args)
  throws Exception {
    ArrayList animalesDomesticos = new ArrayList();
    Class[] tiposAnimalDomestico = {
      AnimalDomestico.class,
      Perro.class,
      Doguillo.class,
      Gato.class,
      Roedor.class,
      Gerbo.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Desplazar en uno para eliminar AnimalDomestico.class:
        int rnd = 1 + (int)(
          Math.random() * (tiposAnimalDomestico.length - 1));
        animalesDomesticos.add(
          tiposAnimalDomestico[rnd].newInstance());
      }
```

```
      } catch(InstantiationException e) {
        System.err.println("No se puede instanciar");
        throw e;
      } catch(IllegalAccessException e) {
        System.err.println("No se puede acceder");
        throw e;
      }
      HashMap h = new HashMap();
      for(int i = 0; i < tiposAnimalDomestico.length; i++)
        h.put(tiposAnimalDomestico[i].toString(),
          new Contador());
      for(int i = 0; i < animalesDomesticos.size(); i++) {
        Object o = animalesDomesticos.get(i);
        // Usando isInstance para eliminar las expresiones
        // instanceof individuales:
        for (int j = 0; j < tiposAnimalDomestico.length; ++j)
          if (tiposAnimalDomestico[j].isInstance(o)) {
            String clave = tiposAnimalDomestico[j].toString();
            ((Contador)h.get(clave)).i++;
          }
      }
      for(int i = 0; i < animalesDomesticos.size(); i++)
        System.out.println(animalesDomesticos.get(i).getClass());
      Iterator clave = h.keySet().iterator();
      while(claves.hasNext()) {
        String nm = (String)claves.next();
        Contador cnt = (Contador)h.get(nm);
        System.out.println(
          nm.substring(nm.lastIndexOf('.') + 1) +
          " cantidad: " + cnt.i);
      }
    }
} ///:~
```

Veamos que el método **isInstance( )** ha eliminado la necesidad de expresiones **instanceof**. Además, esto significa que se pueden añadir nuevos tipos de animal doméstico simplemente cambiando el array **tiposAnimalDomestico**; el resto del programa no necesita modificación alguna (y sí cuando se usaban las expresiones **instanceof**).

instanceof frente a equivalencia de **Class**

Cuando se pregunta por información de tipos, hay una diferencia importante entre cualquier forma de **instanceof** (es decir, **instanceof** o **isInstance( )**, que produce resultados equivalentes) y la comparación directa de los objetos **Class**. He aquí un ejemplo que demuestra la diferencia:

```
//: c12:FamiliaVSTipoExacto.java
// La diferencia entre instanceof y class
```

```
class Base {}
class Derivada extends Base {}

public class FamiliaVsTipoExacto {
  static void comprobar(Object x) {
    System.out.println("Probando x de tipo " +
      x.getClass());
    System.out.println("x instanceof Base " +
      (x instanceof Base));
    System.out.println("x instanceof Derivada " +
      (x instanceof Derivada));
    System.out.println("Base.isInstance(x) " +
      Base.class.isInstance(x));
    System.out.println("Derivada.isInstance(x) " +
      Derivada.class.isInstance(x));
    System.out.println(
      "x.getClass() == Base.class " +
      (x.getClass() == Base.class));
    System.out.println(
      "x.getClass() == Derivada.class " +
      (x.getClass() == Derivada.class));
    System.out.println(
      "x.getClass().equals(Base.class)) " +
      (x.getClass().equals(Base.class)));
    System.out.println(
      "x.getClass().equals(Derivada.class)) " +
      (x.getClass().equals(Derivada.class)));
  }
  public static void main(String[] args) {
    test(new Base());
    test(new Derivada());
  }
} ///:~
```

El método **comprobar( )** lleva a cabo la comprobación de tipos con su argumento usando ambas formas de **instanceof**. Después toma la referencia **Class** y usa == y **equals( )** para probar la igualdad de los objetos **Class**. He aquí la salida:

```
Probando x de tipo class Base
x instanceof Base true
x instanceof Derivada false
Base.isInstance(x) true
Derivada.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derivada.class false
```

```
x.getClass().equals(Base.class)) true
x.getClass.equals(Derivada.class)) false
Probando x de tipo class Derivada
x instanceof Base true
x instanceof Derivada true
Base.isInstance (x) true
Derivada.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derivada.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derivada.class)) true
```

En definitiva, **instanceof** e **isInstance( )** producen exactamente los mismos resultados, al igual que ocurre con **equals( )** y **==**. Pero las pruebas en sí muestran varias conclusiones. En lo que al concepto de tipo se refiere, **instanceof** dice: "¿Eres de esta clase o de una clase derivada de ésta?" Por otro lado, si se comparan los objetos **Class** utilizando ==, no importa la herencia —o es del tipo exacto o no lo es.

Sintaxis RTTI

Java lleva a cabo su RTTI utilizando el objeto **Class**, incluso aunque se esté haciendo alguna conversión. La clase **Class** tiene también otras formas de cara al uso de RTTI.

En primer lugar, hay que conseguir una referencia al objeto **Class** apropiado. Una forma de lograrlo, como se vio en el ejemplo anterior, es usar una cadena de caracteres y el método **Class.forName( )**. Esto es conveniente pues no es necesario un objeto de ese tipo para lograr la referencia **Class.** Sin embargo, si ya se tiene un objeto del tipo en el que se está interesado, se puede lograr la referencia **Class** llamando a un método que sea parte de la clase raíz **Object**: **getClass( )**. Éste devuelve la referencia **Class** que representa al tipo de objeto actual. **Class** tiene muchos métodos interesantes, que se demuestran en el ejemplo siguiente:

```
//: c12:PruebaJuguete.java
// Probando la clase Class.

interface TienePilas {}
interface ResisteAgua {}
interface DisparaCosas {}
class Juguete {
  // Marcar como comentario el siguiente constructor
  // por defecto para ver
  // NoSuchMethodError de (*1*)
  Juguete() {}
  Juguete(int i) {}
}

class JugueteFantasia extends Juguete
```

```
    implements TienePilas,
      ResisteAgua, DisparaCosas {
  JugueteFantasia() { super(1); }
}

public class PruebaJuguete {
  public static void main(String[] args)
  throws Exception {
    Class c = null;
    try {
      c = Class.forName("JugueteFantasia");
    } catch(ClassNotFoundException e) {
      System.err.println("No se puede encontrar JugueteFantasia");
      throw e;
    }
    imprimirInfo(c);
    Class[] semblantes = c.getInterfaces();
    for(int i = 0; i < semblantes.length; i++)
      imprimirInfo(semblantes[i]);
    Class cy = c.getSuperclass();
    Object o = null;
    try {
      // Requiere del constructor por defecto:
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {
      System.err.println("No se puede instanciar");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("No se puede acceder");
      throw e;
    }
    imprimirInfo(o.getClass());
  }
  static void imprimirInfo(Class cc) {
    System.out.println(
      "Nombre de clase: " + cc.getName() +
      " ¿es interfaz? [" +
      cc.isInterface() + "]");
  }
} ///:~
```

Veamos que la **class JugueteFantasia** es bastante complicada, puesto que hereda de **Juguete** e **implementa** las **interfaces TienePilas**, **ResisteAgua** y **DisparaCosas**. En el método **main( )**, se crea una referencia **Class** y se inicializa a la **Class JugueteFantasia** usando **forName( )** dentro de un bloque **try** apropiado.

El método **Class.getInterfaces( )** devuelve un array de objetos **Class** que representa las interfaces de interés contenidas en el objeto **Class**.

Si se tiene un objeto **Class** también se puede pedir su clase directa base utilizando **getSuperclass ( )**. Esto, por supuesto, devuelve una referencia **Class** por la que se puede preguntar más adelante. Esto significa que, en tiempo de ejecución, se puede descubrir toda la jerarquía de clases de un objeto.

El método **newInstance( )** de **Class** puede, en primer lugar, parecer justo otra manera de **clone( )** (clonar) un objeto. Sin embargo, se puede crear un objeto nuevo con **newInstance( )** *sin* un objeto existente, como se ha visto, porque no hay objeto **Juguete** —sólo **cy**, que es una referencia al objeto **Class** de **y**. Ésta es una forma de implementar un "constructor virtual", que te permite decir: "No sé exactamente de qué tipo eres, pero de todas formas créate a ti mismo de la forma adecuada". En el ejemplo de arriba, **cy** es simplemente una referencia **Class**, sin que se conozca más información sobre su tipo en tiempo de compilación. Y cuando se crea una instancia nueva, se obtiene una **referencia Object**. Pero esa referencia apunta a un objeto **Juguete**. Por supuesto, antes de poder enviar cualquier mensaje que no sea aceptado por **Object**, hay que investigar esta referencia un poco y hacer algún tipo de conversión. Además, la clase que se está creando con **newInstance( )** debe tener un constructor por defecto. En la sección siguiente se verá cómo crear objetos de clases dinámicamente utilizando cualquier constructor, con el API Java *reflectivo*.

El método final del listado es **imprimirInfo( )**, que toma una referencia **Class** y consigue su nombre con **getName( )**, y averigua si se trata o no de una interfaz con **isInterface( )**.

La salida de este programa es:

```
Class name: JugueteFantasia ¿es interfaz? [false]
Class name: TienePilas ¿es interfaz? [true]
Class name: ResisteAgua ¿es interfaz? [true]
Class name: DisparaCosas ¿es interfaz? [true]
Class name: Juguete ¿es interfaz? [false]
```

Por consiguiente, con el objeto **Class** se puede averiguar casi todo lo que se desee saber sobre un objeto.

Reflectividad: información de clases en tiempo de ejecución

Si no se sabe el tipo exacto de un objeto, RTTI te lo dice. Sin embargo, hay una limitación: debe conocerse el tipo en tiempo de compilación para poder detectarlo usando RTTI y hacer algo útil con la información. Dicho de otra forma, el compilador debe conocer información sobre todas las clases con las que trabaja de cara a la RTTI.

Esto no parece una gran limitación a primera vista, pero supóngase que se obtiene una referencia a un objeto que no está en el espacio del programa. De hecho, la clase del objeto ni siquiera está disponible para el programa en tiempo de compilación. Por ejemplo, supóngase que se obtiene un conjunto de bytes de un archivo de disco o de una conexión de red, y se sabe que se trata de una cla-

se. Puesto que el compilador no puede saber nada acerca de la clase mientras está compilando el código, ¿cómo podría llegar a usarla?

En los entornos de programación tradicionales éste parece un escenario poco probable. Pero a medida que nos introducimos en un mundo de programación mayor, hay casos importantes en los que esto ocurre. El primero es la programación basada en componentes, en la que se construyen proyectos utilizando *Rapid Application Development*[2] (RAD) en una herramienta para la construcción de aplicaciones. Se trata de enfoques visuales para crear programas (que se muestran en pantallas como "formularios") moviendo iconos que representan componentes dentro de los formularios. Estos componentes se configuran después estableciendo algunos de los valores en tiempo de programación. Esta configuración en tiempo de diseño exige que todos los componentes sean instanciables, que expongan partes de sí mismos, y que permitan la lectura y asignación de valores. Además, los componentes que manejen eventos de la IGU deben exponer información sobre los métodos apropiados de forma que el entorno RAD pueda ayudar al programador a superponer estos métodos de manejo de eventos. La reflectividad proporciona el mecanismo para detectar los métodos disponibles y producir los nombres de método. Java proporciona una estructura para programación basada en componentes a través de los denominados JavaBeans (descritos en el Capítulo 13).

Otra motivación que conduce al descubrimiento de información de clases en tiempo de ejecución es proporcionar la habilidad de crear y ejecutar objetos en plataformas remotas a través de la red. A esto se le llama *Remote Method Invocation*[3] (RMI) y permite a un programa Java tener objetos distribuidos por varias máquinas. Esta distribución puede darse por varias razones: por ejemplo, quizás se está haciendo una tarea de computación intensiva y se desea dividirla entre varias máquinas ociosas para acelerar el rendimiento global. En ocasiones se podría desear ubicar código que maneje tipos de tareas particulares (por ejemplo, "Reglas de negocio" en una arquitectura cliente/servidor multicapa) en una determinada máquina, de forma que esa máquina se convierte en un repositorio común describiendo esas acciones, y que puede modificarse sencillamente para afectar a todo el sistema. (¡Se trata de un desarrollo interesante, pues la máquina sólo existe para facilitar los cambios en el código!) Además de esto, la computación distribuida también permite soportar hardware especializado que puede ser necesario para alguna tarea en particular —inversión de matrices, por ejemplo— pero inapropiado o demasiado caro para programación de propósito general.

La clase **Class** (descrita previamente en este capítulo) soporta el concepto de *reflectividad*, y hay una biblioteca adicional, **java.lang.reflect**, con las clases **Field**, **Method** y **Constructor** (cada una implementa el **interfaz Member**). Los objetos de este tipo los crea la JVM en tiempo de ejecución para representar el miembro correspondiente de clase desconocida. Se pueden usar después los **Constructor**s para crear nuevos objetos, los métodos **get( )** y **set( )** para leer y modificar los campos asociados con los objetos **Field**, y el método **invoke( )** para llamar al método asociado con un objeto **Method**. Además, se puede invocar a los métodos **getFields( )**, **getMethods( )**, **getConstructors( )**, etc. para devolver arrays de objetos que representen los campos, métodos y constructores. (Se puede averiguar aún más buscando la clase **Class** en la documentación en línea.)

---

[2] N. del traductor: Desarrollo Rápido de Aplicaciones.

[3] N. del traductor: Invocación de Métodos Remotos.

Por consiguiente, se puede determinar completamente en tiempo de ejecución la información de clase de los objetos anónimos, sin tener que saber nada en tiempo de compilación.

Es importante darse cuenta de que no hay nada mágico en la reflectividad. Cuando se usa la reflectividad para interactuar con un objeto de un tipo desconocido, la JVM simplemente mira al objeto y ve que pertenece a una clase particular (como el RTTI ordinario) pero en ese momento, antes de hacer nada más, se debe cargar el objeto **Class**. Por consiguiente, debe estar disponible para la JVM el archivo **.class** de ese tipo particular, bien en la máquina local o bien a través de la red. Por tanto, la verdadera diferencia entre RTTI y la reflectividad es que con la RTTI el compilador abre y examina el fichero **.class** en tiempo de compilación. Dicho de otra forma, se puede llamar a todos los métodos del objeto de forma "normal". Con la reflectividad, el archivo **.class** no está disponible en tiempo de compilación; se abre y examina por el entorno en tiempo de ejecución.

Un extractor de métodos de clases

Las herramientas de reflectividad se usarán directamente muy pocas veces; están en el lenguaje para dar soporte a otras facetas de Java, como la serialización de objetos (Capítulo 11), JavaBeans (Capítulo 13) y RMI (Capítulo 15). Sin embargo, hay veces en las que es bastante útil ser capaz de extraer dinámicamente información sobre una clase. Una herramienta extremadamente útil es el extractor de métodos de clases. Como se mencionó anteriormente, mirar el código fuente de una definición de clase o la documentación en línea sólo muestra los métodos definidos o superpuestos *dentro de esa definición de clase*. Pero podría haber otras muchas docenas disponibles provenientes de clases base. Localizarlos es tedioso y encima consume mucho tiempo[4]. Afortunadamente, la reflectividad proporciona una forma de escribir, una herramienta sencilla que mostrará automáticamente todo la interfaz. Funciona así:

```
//: c12:MostrarMetodos.java
// Usando la reflectividad para mostrar todos los métodos
// de una clase, incluso los definidos en
// la clase base.
import java.lang.reflect.*;

public class MostrarMetodos {
  static final String uso =
    "uso: \n" +
    "MostrarMetodos nombre.clase.calificado\n" +
    "Para mostrar todos los metodos de la clase o: \n" +
    "MostrarMetodos nombre.clase.calificado palabra\n" +
    "Para buscar todos los metodos que involucran a 'palabra'";
  public static void main(String[] args) {
    if(args.length < 1) {
```

[4] Especialmente en el pasado. Sin embargo, Sun ha mejorado mucho su documentación HTML de Java de forma que es más fácil acceder a los métodos de la clase base.

```
        System.out.println(uso);
        System.exit(0);
      }
      try {
        Class c = Class.forName(args[0]);
        Method[] m = c.getMethods();
        Constructor[] ctor = c.getConstructors();
        if(args.length == 1) {
          for (int i = 0; i < m.length; i++)
            System.out.println(m[i]);
          for (int i = 0; i < ctor.length; i++)
            System.out.println(ctor[i]);
        } else {
          for (int i = 0; i < m.length; i++)
            if(m[i].toString()
                .indexOf(args[1])!= -1)
              System.out.println(m[i]);
          for (int i = 0; i < ctor.length; i++)
            if(ctor[i].toString()
                .indexOf(args[1])!= -1)
            System.out.println(ctor[i]);
        }
      } catch(ClassNotFoundException e) {
        System.err.println("Clase inexistente: " + e);
      }
    }
} ///:~
```

Los métodos de **Class**, **getMethod( )** y **getConstructors( )** devuelven un array de **Method** y **Constructor** respectivamente. Cada una de estas clases tiene más métodos para diseccionar los nombres, parámetros y valores de retorno de los métodos que representan. Pero también se puede usar **toString( )**, como en ese caso, para producir un **String** con la signatura completa del método. El resto del código simplemente sirve para extraer información de línea de comandos, determinar si una signatura en particular coincide con la cadena de caracteres destino (utilizando **indexOf( )**), e imprimir los resultados.

Esto muestra la reflectividad en acción, puesto que no se puede conocer el resultado producido por **Class.forName( )** en tiempo de compilación, y por tanto se extrae toda la información de signatura de métodos en tiempo de ejecución. Si se investiga la documentación *en línea* relativa a la reflectividad, se ve que es suficiente para establecer y construir una llamada a un objeto totalmente conocido en tiempo de compilación (habrá algunos ejemplos de esto al final del presente libro). De nuevo, puede que uno nunca necesite hacer esto —este soporte está por RMI y para que un entorno de programación pueda soportar JavaBeans— pero es interesante.

Un experimento interesante es ejecutar

```
java MostrarMetodos MostrarMetodos
```

Esta invocación produce un listado que incluye un constructor por defecto **public**, incluso aunque se pueda ver en el código que no se definió ningún constructor. El constructor que se ve es el que ha sido automáticamente sintetizado por el compilador. Si después se convierte **MostrarMetodos** en una clase no **public** (es decir, amiga), el constructor sintetizado por defecto deja de mostrar la salida. Al constructor por defecto sintetizado se le da automáticamente el mismo acceso que a la clase.

La salida de **MostrarMetodos** sigue siendo algo tediosa. He aquí, por ejemplo, una porción de la salida producida al invocar **java MostrarMetodos java.lang.String**:

```
public boolean
  java.lang.String.startsWith(java.lang.String,int)
public boolean
  java.lang.String.startsWith(java.lang.String)
public boolean
  java.lang.String.endsWith(java.lang.String)
```

Sería incluso mejor si se eliminaran los calificadores del estilo de **java.lang**. La clase **StreamTokenizer** presentada en el capítulo anterior puede ayudar a crear una herramienta que solucione este problema:

```
//: com:bruceeckel:util:EliminarCalificadores.java
package com.bruceeckel.util;
import java.io.*;

public class EliminarCalificadores {
  private StreamTokenizer st;
  public EliminarCalificadores(String calificado) {
    st = new StreamTokenizer(
      new StringReader(calificado));
    st.ordinaryChar(' '); // Mantener los espacios
  }
  public String obtenerSiguiente() {
    String s = null;
    try {
      int simbolo = st.nextToken();
      if(simbolo != StreamTokenizer.TT_EOF) {
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = null;
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = new String(st.sval);
```

```
                break;
             default: // único carácter en ttype
                s = String.valueOf((char)st.ttype);
          }
       }
    } catch(IOException e) {
       System.err.println("Error recuperando simbolo");
    }
    return s;
  }
  public static String eliminar(String calificado) {
    EliminarCalificadores ec =
      new EliminarCalificadores(calificado);
    String s = "", si;
    while((si = ec.obtenersiguiente()) != null) {
      int ultimoPunto = si.lastIndexOf('.');
      if(ultimoPunto != -1)
        si = si.substring(ultimoPunto + 1);
      s += si;
    }
    return s;
  }
} ///:~
```

Para facilitar la reutilización, esta clase está ubicada en **com.bruceeckel.util**. Como puede verse, hace uso de **StreamTokenizer** y la manipulación de **String**s para hacer su trabajo.

La versión nueva del programa usa la clase de arriba para limpiar la salida:

```
//: c12:LimpiarMostrarMetodos.java
// Mostrar Métodos sin calificadores
// para que los resultados sean más fáciles
// de leer.
import java.lang.reflect.*;
import com.bruceeckel.util.*;

public class LimpiarMostrarMetodos {
  static final String uso =
    "uso: \n" +
    "LimpiarMostrarMetodos nombre.clase.calificado\n" +
    "Para mostrar todos los metodos de la clase o: \n" +
    "LimpiarMostrarMetodos nombre.clase.calificado palabra\n" +
    "Para buscar todos los metodos que involucran a 'palabra'";
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(uso);
```

```
            System.exit(0);
        }
        try {
            Class c = Class.forName(args[0]);
            Method[] m = c.getMethods();
            Constructor[] ctor = c.getConstructors();
            // Convertirlo en un array de Strings limpios:
            String[] n =
                new String[m.length + ctor.length];
            for(int i = 0; i < m.length; i++) {
                String s = m[i].toString();
                n[i] = EliminarCalificadores.eliminar(s);
            }
            for(int i = 0; i < ctor.length; i++) {
                String s = ctor[i].toString();
                n[i + m.length] =
                    EliminarCalificadores.eliminar(s);
            }
            if(args.length == 1)
                for (int i = 0; i < n.length; i++)
                    System.out.println(n[i]);
            else
                for (int i = 0; i < n.length; i++)
                    if(n[i].indexOf(args[1])!= -1)
                        System.out.println(n[i]);
        } catch(ClassNotFoundException e) {
            System.err.println("Clase inexistente: " + e);
        }
    }
} ///:~
```

La clase **LimpiarMostrarMetodos** es bastante similar a la **MostrarMetodos** previa, excepto en que toma los arrays de **Method** y **Constructor** y los convierte en un único array de **String**s. Cada uno de estos objetos **String** se pasa después a través de **EliminarCalificadores.eliminar( )** para eliminar toda la cualificación de métodos.

Esta herramienta puede ser un verdadero ahorro de tiempo al programar, en las ocasiones en que no se puede recordar si una clase tiene un método en particular y no se desea ir recorriendo toda la jerarquía de clases en la documentación en línea, o si se desconoce si la clase puede hacer algo, por ejemplo, con objetos **Color**.

El Capítulo 13 contiene una versión IGU de este programa (personalizada para extraer información para componentes Swing) de forma que se puede dejar que se ejecute mientras se escribe el código para permitir búsquedas rápidas.

Resumen

RTTI permite descubrir información de tipos desde una referencia a una clase base anónima. Por consiguiente, es muy posible que sea mal utilizada por un novato, puesto que podría cobrar sentido antes de lo que lo cobran los métodos polimórficos. Para mucha gente proveniente de un trasfondo procedural, es difícil no organizar sus programas en conjuntos de sentencias **switch**. Podrían lograr esto con RTTI, y por consiguiente perder el valor importante del polimorfismo en el desarrollo y mantenimiento de código. La intención de java es que se usen llamadas a métodos polimórficos a través del código, y usar RTTI sólo cuando se deba.

Sin embargo, el uso de llamadas a métodos polimórficos como se pretende requiere de un control de la definición de la clase base porque en algún momento de la extensión del programa se podría descubrir que la clase base no implementa el método que se necesita. Si la clase base proviene de una biblioteca o está controlada de alguna forma por alguien más, una solución al problema sería la RTTI: se puede heredar un nuevo tipo y añadir el método extra. En cualquier otro lugar del código es posible detectar el tipo particular e invocar a ese método en especial. Esto no destruye el polimorfismo y la extensibilidad del programa porque la adición de un nuevo tipo no exigirá buscar sentencias switch por todo el programa. Sin embargo, cuando se añade código nuevo en el cuerpo principal que requiera una nueva faceta, hay que usar RTTI para detectar el tipo en particular.

Poner una característica en la clase base podría significar que, en beneficio de una clase particular, todas las otras clases derivadas de esa base requieran algún fragmento insignificante de un método. Esto hace la interfaz menos limpia, y molesta a aquéllos que deben superponer métodos abstractos cuando se derivan de esa clase base. Por ejemplo, considérese una jerarquía de clases que represente los instrumentos musicales. Supóngase que se desea limpiar las válvulas de soplado de todos los instrumentos de la orquesta que las tengan. Una opción sería poner un método **limpiarValvulaSoplado( )** en la clase base **Instrumento**, pero esto es confuso pues implicaría que los instrumentos de **Percusión** y **Electrónico**s también tuvieran válvulas de soplado. RTTI proporciona una solución mucho más razonable porque se puede ubicar el método en la clase específica (en este caso **Viento**), donde es apropiado. Sin embargo, una solución más adecuada es poner un método **prepararInstrumento( )** en la clase base, pero podría no verse cuando se solucione el problema por primera vez y podría asumir erróneamente que hay que usar RTTI.

Finalmente, RTTI solucionará algunos problemas de eficiencia. Si el código hace uso elegantemente del polimorfismo, pero resulta que uno de los objetos reacciona a este código de propósito general de forma horriblemente ineficiente, se puede extraer este tipo usando RTTI y escribir código específico del caso para mejorar la eficiencia. Sin embargo, hay que ser cauto y no buscar la eficiencia demasiado pronto. Es una trampa seductora. Es mejor hacer *primero* que el programa funcione, y decidir después si se ejecuta lo suficientemente rápido, y sólo en ese momento enfrentarse a aspectos de eficiencia.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Añadir **Romboide** a **Figuras.java**. Crear un **Romboide**, aplicar un molde hacia arriba a **Figura**, y después hacer una conversión de vuelta hacia un **Romboide**. Intentar aplicar un molde hacia abajo a **Círculo** y ver qué pasa.

2. Modificar el Ejercicio 1 de forma que use **instanceof** para comprobar el tipo antes de hacer la conversión hacia abajo.

3. Modificar **Figuras.java** de forma que "resalte" (ponga un *flag*) en todos los polígonos de un tipo en particular. El método **toString( )** de cada **Figura** derivado debería indicar si esa **Figura** está "resaltada".

4. Modificar **Confiteria.java** de forma que se controle la creación de cada tipo de objeto por un parámetro de línea de comandos. Es decir, si la línea de comandos es "**java Confiteria Caramelo**", que sólo se cree el objeto **Caramelo**. Darse cuenta de cómo es posible controlar los objetos **Class** que se crean vía la línea de comandos.

5. Añadir un nuevo tipo de **AnimalDomestico** a **RecuentoAnimalDomestico3.java**. Verificar que se crea y que cuenta correctamente en el método **main( )**.

6. Escribir un método que tome un objeto e imprima recursivamente todas las clases en su jerarquía de objetos.

7. Modificar el Ejercicio 6 de forma que use **Class.getDeclaredFields( )** para mostrar también información de los campos de una clase.

8. En **PruebaJuguete.java**. marcar como comentario el constructor por defecto de **Juguete** y explicar lo que ocurre.

9. Incorporar un nuevo tipo de interfaz a **PruebaJuguete.java** y verificar que se detecta y muestra correctamente.

10. Crear un nuevo tipo de contenedor que use un **private ArrayList** para guardar los objetos. Capturar el tipo del primer tipo que se introduce en él y permitir al usuario insertar objetos sólo de ese tipo a partir de ese momento.

11. Escribir un programa para determinar si un array de **char** es un tipo primitivo o un objeto auténtico.

12. Implementar **limpiarValvulaSoplado( )** como se describe en el resumen.

13. Implementar el método **rotar(Figura)** descrito en este capítulo de forma que compruebe si está rotando un **Círculo** (y si es el caso, que no lleve a cabo la operación).

14. Modificar el Ejercicio 6 de forma que use la reflectividad en vez de RTTI.

15. Modificar el Ejercicio 7 de forma que use la reflectividad en vez de RTTI.

16. En **PruebaJuguete.java**, utilizar la reflectividad para crear un objeto **Juguete** usando un constructor distinto del constructor por defecto.

17. Buscar la interfaz de **java.lang.Class** en la documentación HTML de Java que hay en *http://java.sun.com*. Escribir un programa que tome el nombre de una clase como parámetro de línea de comandos, y después use los métodos **Class** para volcar toda la información disponible para esa clase. Probar el programa con una biblioteca estándar y una clase creada por uno mismo.

13: Crear ventanas y applets

Una guía de diseño fundamental es "haz las cosas simples de forma sencilla, y las cosas difíciles hazlas posibles."[1]

La meta original de diseño de la biblioteca de interfaz gráfico de usuario (IGU) en Java 1.0 era permitir al programador construir un IGU que tuviera buen aspecto en todas las plataformas. Esa meta no se logró. En su lugar, el *Abstract Window Toolkit* (AWT) de Java 1.0 produce un IGU que tiene una apariencia igualmente mediocre en todos los sistemas. Además, es restrictivo: sólo se pueden usar cuatro fuentes y no se puede acceder a ninguno de los elementos de IGU más sofisticados que existen en el sistema operativo. El modelo de programación de AWT de Java 1.0 también es siniestro y no es orientado a objetos. En uno de mis seminarios, un estudiante (que había estado en Sun durante la creación de Java) explicó el porqué de esto: el AWT original había sido conceptualizado, diseñado e implementado en un mes. Ciertamente es una maravilla de la productividad, además de una lección de por qué el diseño es importante.

La situación mejoró con el modelo de eventos del AWT de Java 1.1, que toma un enfoque orientado a objetos mucho más claro, junto con la adición de JavaBeans, un modelo de programación basado en componentes orientado hacia la creación sencilla de entornos de programación visuales. Java 2 acaba esta transformación alejándose del AWT de Java 1.0 esencialmente, reemplazando todo con las *Java Foundation Classes* (JFC), cuya parte IGU se denomina "Swing". Se trata de un conjunto de JavaBeans fáciles de usar, y fáciles de entender que pueden ser arrastrados y depositados (además de programados a mano) para crear un IGU con el que uno se encuentre finalmente satisfecho. La regla de la "revisión 3" de la industria del software (un producto no es bueno hasta su tercera revisión) parece cumplirse también para los lenguajes de programación.

Este capítulo no cubre toda la moderna biblioteca Swing de Java 2, y asume razonablemente que Swing es la biblioteca IGU destino final de Java. Si por alguna razón fuera necesario hacer uso del "viejo" AWT (porque se está intentando dar soporte a código antiguo o se tienen limitaciones impuestas por el navegador), es posible hacer uso de la introducción que había en la primera edición de este libro, descargable de *http://www.BruceEckel.com* (incluida también en el CD ROM que se adjunta a este libro).

Al principio de este capítulo veremos cómo las cosas son distintas si se trata de crear un *applet* o si se trata de crear una aplicación ordinaria haciendo uso de Swing, y cómo crear programas que son tanto *applets* como aplicaciones, de forma que se pueden ejecutar bien dentro de un *browser*, o bien desde línea de comandos. Casi todos los ejemplos IGU de este libro serán ejecutables bien como *applet*, o como aplicaciones.

---

[1] Hay una variación de este dicho que se llama "el principio de asombrarse al mínimo", que dice en su esencia: "No sorprenda al usuario".

Hay que ser conscientes de que este capítulo no es un vocabulario exhaustivo de los componentes Swing o de los métodos de las clases descritas. Todo lo que aquí se presenta es simple a propósito. La biblioteca Swing es vasta y la meta de este capítulo es sólo introducirnos con la parte esencial y más agradable de los conceptos. Si se desea hacer más, Swing puede proporcionar lo que uno desee siempre que uno se enfrente a investigarlo.

Asumimos que se ha descargado e instalado la documentación HTML de las bibliotecas de Java (que es gratis) de *http://java.sun.com* y que se navegará a lo largo de las clases **javax.swing** de esa documentación para ver los detalles completos y los métodos de la biblioteca Swing. Debido a la simplicidad del diseño Swing, generalmente esto será suficiente información para solucionar todos los problemas. Hay numerosos libros (y bastante voluminosos) dedicados exclusivamente a Swing y son altamente recomendables si se desea mayor nivel de profundidad, o cuando se desee cambiar el comportamiento por defecto de la biblioteca Swing.

A medida que se aprendan más cosas sobre Swing se descubrirá que:

1. Swing es un modelo de programación mucho mejor que lo que se haya visto probablemente en otros lenguajes y entornos de desarrollo. Los JavaBeans (que no se presentarán hasta el final de este capítulo) constituyen el marco de esa biblioteca.

2. Los "constructores IGU" (entornos de programación visual) son un aspecto *de rigueur* de un entorno de desarrollo Java completo. Los JavaBeans y Swing permiten al constructor IGU escribir código por nosotros a medida que se ubican componentes dentro de formularios utilizando herramientas gráficas. Esto no sólo acelera rápidamente el desarrollo utilizando construcción IGU, sino que permite un nivel de experimentación mayor, y por consiguiente la habilidad de probar más diseños y acabar generalmente con uno mejor.

3. La simplicidad y la tan bien diseñada naturaleza de Swing significan que incluso si se usa un constructor IGU en vez de codificar a mano, el código resultante será más completo —esto soluciona un gran problema con los constructores IGU del pasado, que podían generar de forma sencilla código ilegible.

Swing contiene todos los componentes que uno espera ver en un IU moderno, desde botones con dibujos hasta árboles y tablas. Es una gran biblioteca, pero está diseñada para tener la complejidad apropiada para la tarea a realizar —es algo simple, no hay que escribir mucho código, pero a medida que se intentan hacer cosas más complejas, el código se vuelve proporcionalmente más complejo. Esto significa que nos encontramos ante un punto de entrada sencillo, pero se tiene a mano toda la potencia necesaria.

A mucho de lo que a uno le gustaría de Swing se le podría denominar "ortogonalidad de uso". Es decir, una vez que se captan las ideas generales de la biblioteca, se pueden aplicar en todas partes. En primer lugar, gracias a las convenciones estándar de denominación, muchas de las veces, mientras se escriben estos ejemplos, se pueden adivinar los nombres de los métodos y hacerlo bien a la primera, sin investigar nada más. Ciertamente éste es el sello de un buen diseño de biblioteca. Además, generalmente se pueden insertar componentes a los componentes ya existentes de forma que todo funcione correctamente.

En lo que a velocidad se refiere, todos los componentes son "ligeros", y Swing se ha escrito completamente en Java con el propósito de lograr portabilidad.

La navegación mediante teclado es automática —se puede ejecutar una aplicación Java sin usar el ratón, y no es necesaria programación extra. También se soporta desplazamiento sin esfuerzo —simplemente se envuelven los componentes en un **JScrollPane** a medida que se añaden al formulario. Aspectos como etiquetas de aviso simplemente requieren de una línea de código.

Swing también soporta una faceta bastante más radical denominada "*apariencia* conectable" que significa que se puede cambiar dinámicamente la apariencia del IU para que se adapte a las expectativas de los usuarios que trabajen en plataformas y sistemas operativos distintos. Incluso es posible (aunque difícil) inventar una apariencia propia.

El applet básico

Una de las metas de diseño de Java es la creación de *applets*, que son pequeños programas que se ejecutan dentro del navegador web. Dado que tienen que ser seguros, se limitan a lo que pueden lograr. Sin embargo, los *applets* son una herramienta potente que soporta programación en el lado cliente, uno de los aspectos fundamentales de la Web.

Restricciones de applets

La programación dentro de un *applet* es tan restrictiva que a menudo se dice que se está "dentro de una caja de arena" puesto que siempre se tiene a alguien —es decir, el sistema de seguridad de tiempo de ejecución de Java— vigilando.

Sin embargo, uno también se puede salir de la caja de arena y escribir aplicaciones normales en vez de *applets*, en cuyo caso se puede acceder a otras facetas del S.O. Hemos estado escribiendo aplicaciones normales a lo largo de todo este libro, pero han sido *aplicaciones de consola* sin componentes gráficos. También se puede usar Swing para construir interfaces IGU para aplicaciones normales.

Generalmente se puede responder a la pregunta de qué es lo que un *applet* puede hacer mirando a lo que *se supone* que hace: extender la funcionalidad de una página web dentro de un navegador. Puesto que, como navegador de la Red, nunca se sabe si una página web proviene de un sitio amigo o no, se desea que todo código que se ejecute sea seguro. Por tanto, las mayores restricciones que hay que tener en cuenta son probablemente:

1. *Un applet no puede tocar el disco local.* Esto significa escribir *o* leer, puesto que no se desearía que un *applet* pudiera leer y transmitir información privada a través de Internet sin permiso. Se evita la escritura, por supuesto, dado que supondría una invitación abierta a los virus. Java ofrece *firmas digitales* para *applets*. Muchas restricciones de *applets* se suavizan cuando se elige la ejecución de *applets de confianza* (los firmados por una fuente de confianza) para tener acceso a la máquina.

2. *Puede llevar más tiempo mostrar los applets*, puesto que hay que descargarlos por completo cada vez, incluyendo una solicitud distinta al servidor por cada clase diferente. El navegador puede introducir el *applet* en la caché, pero no hay ninguna garantía de que esto ocurra. Debido a esto, probablemente habría que empaquetar los *applets* en un JAR (Java ARchivo) que combina todos los componentes del *applet* (incluso otros archivos **.class** además de imágenes y sonidos) dentro de un único archivo comprimido que puede descargarse en una única transacción servidora. El "firmado digital" está disponible para cada entrada digital de un archivo JAR.

Ventajas de los *applets*

Si se puede vivir con las restricciones, los *applets* tienen también ventajas significativas cuando se construyen aplicaciones cliente/servidor u otras aplicaciones en red:

1. *No hay aspectos de instalación*. Un *applet* tiene independencia completa de la plataforma (incluyendo la habilidad de reproducir sin problemas archivos de sonido, etc.) por lo que no hay que hacer ningún cambio en el código en función de la plataforma ni hay que llevar a cabo ninguna "adaptación" en el momento de la instalación. De hecho, la instalación es automática cada vez que el usuario carga la página web que contiene *applets*, de forma que las actualizaciones se darán de forma inadvertida y automáticamente. En los sistemas cliente/servidor tradicionales, construir e instalar nuevas versiones del software cliente es siempre una auténtica pesadilla.

2. *No hay que preocuparse de que el código erróneo cause ningún mal a los sistemas de alguien*, gracias a la propia seguridad implícita en la esencia de Java y en la estructura de los *applets*. Esto, junto con el punto anterior, convierte a Java en un lenguaje popular para las denominadas aplicaciones cliente/servidor de *intranet* que sólo residen dentro de un campo de operación restringido dentro de una compañía donde se puede especificar y/o controlar el entorno del usuario (el navegador web y sus añadidos).

Debido a que los *applets* se integran automáticamente con el HTML, hay que incluir un sistema de documentación embebido independiente de la plataforma para dar soporte al *applet*. Se trata de un problema interesante, puesto que uno suele estar acostumbrado a que la documentación sea parte del programa en vez de al revés.

Marcos de trabajo de aplicación

Las bibliotecas se agrupan generalmente dependiendo de su funcionalidad. Algunas bibliotecas, por ejemplo, se usan como tales, independientemente. Las clases **String** y **ArrayList** de la biblioteca estándar de Java son ejemplos de esto. Otras bibliotecas están diseñadas específicamente como bloques que permiten construir otras clases. Cierta categoría de biblioteca es el *marco de trabajo de aplicación*, cuya meta es ayudar en la construcción de aplicaciones proporcionando una clase o un conjunto de clases que producen el comportamiento básico deseado para toda aplicación de un tipo particular. Posteriormente, para adaptar el comportamiento a nuestras propias necesidades, hay que heredar de la clase aplicación y superponer los métodos que interesen. El marco de trabajo de apli-

cación es un buen ejemplo de "separar las cosas que cambian de las que permanecen invariables", puesto que intenta localizar todas las partes únicas de un programa en los métodos superpuestos[2].

Los *applets* se construyen utilizando un marco de trabajo de aplicación. Se hereda de **JApplet** y se superponen los métodos apropiados. Hay unos pocos métodos que controlan la creación y ejecución de un *applet* en una página web:

Método	Operación
init( )	Se invoca automáticamente para lograr la primera inicialización del *applet*, incluyendo la disposición de los componentes. Este método siempre se superpone.
start( )	Se invoca cada vez que se visualiza un *applet* en el navegador para permitirle empezar sus operaciones normales (especialmente las que se apagan con **stop( )**).También se invoca tras **init( )**.
stop( )	Se invoca cada vez que un *applet* se aparta de la vista de un navegador web para permitir al *applet* apagar operaciones caras. Se invoca también inmediatamente antes de **destroy( )**.
destroy( )	Se invoca cada vez que se está descargando un *applet* de una página para llevar a cabo la liberación final de recursos cuando se deja de usar el *applet*.

Con esta información ya se puede crear un *applet* simple:

```
//: c13:Applet1.java
// Un applet muy simple.
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("¡Applet!"));
  }
} ///:~
```

Nótese que no se exige a los *applets* tener un método **main( )**. Todo se incluye en el marco de trabajo de aplicación; el código de arranque se pone en el **init( )**.

En este programa, la única actividad que se hace es poner una etiqueta de texto en el *applet*, vía la clase **JLabel** (la vieja AWT se apropió del nombre **Label** además de otros nombres de componen-

[2] Éste es un ejemplo del patrón de diseño denominado *método plantilla*.

tes, por lo que es habitual ver que los componentes Swing empiezan por una "J"). El constructor de esta clase toma un **String** y lo usa para crear la etiqueta. En el programa de arriba se coloca esta etiqueta en el formulario.

El método **init( )** es el responsable de poner todos los componentes en el formulario haciendo uso del método **add( )**. Se podría pensar que debería ser posible invocar simplemente a **add( )** por sí mismo, y de hecho así solía ser en el antiguo AWT. Sin embargo, Swing requiere que se añadan todos los componentes al "panel de contenido" de un formulario, y por tanto hay que invocar a **getContentPane( )** como parte del proceso **add( )**.

Ejecutar applets dentro de un navegador web

Para ejecutar este programa, hay que ubicarlo dentro de una página web y ver esa página dentro de un navegador web con Java habilitado. Para ubicar un *applet* dentro de una página web se pone una etiqueta especial dentro de la fuente HTML de esa página web para indicar a la misma[3] cómo cargar y ejecutar el *applet*.

Este proceso era muy simple cuando Java en sí era simple y todo el mundo estaba en la misma línea e incorporaba el mismo soporte Java en sus navegadores web. Se podría haber continuado simplemente con un fragmento muy pequeño de código HTML dentro de la página web, así:

```
<applet code=Applet1 width=100 height=50>
</applet>
```

Después vinieron las guerras de navegadores y lenguajes y perdimos nosotros (los programadores y los usuarios finales). Tras cierto periodo de tiempo, JavaSoft se dio cuenta de que no podía esperar a que los navegadores siguieran soportando el buen gusto de Java, y la única solución era proporcionar algún tipo de añadido que se relacionara con el mecanismo de extensión del navegador. Utilizando el mecanismo de extensión (que los vendedores de navegadores no pueden deshabilitar —en un intento de ganar ventaja competitiva— sin romper con todas las extensiones de terceros), JavaSoft garantiza que un vendedor antagonista no pueda arrancar Java de su navegador web.

Con Internet Explorer, el mecanismo de extensión es el control ActiveX, y con Netscape, los *plugins*. He aquí el aspecto que tiene la página HTML más sencilla para **Applet1:**[4]

```
//:! c13:Applet1.html
<html><head><title>Applet1</title></head><hr>
<OBJECT
  classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  width="100" height="50" align="baseline"
```

[3] Se asume que el lector está familiarizado con lo básico de HTML. No es demasiado difícil, y hay cientos de libros y otros recursos.

[4] Esta página —en particular la porción "clsid"— parecía funcionar bien tanto con JDK 1.2.2 como JDK 1.3 rc-1. Sin embargo, puede ser que en el futuro haya que cambiar algo la etiqueta. Pueden encontrar detalles al respecto en *java.sun.com*.

```
codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-1_2_2-
win.cab#Version=1,2,2,0">
<PARAM NAME="code" VALUE="Applet1.class">
<PARAM NAME="codebase" VALUE=".">
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2.2">
<COMMENT>
  <EMBED type=
     "application/x-java-applet;version=1.2.2"
     width="200" height="200" align="baseline"
     code="Applet1.class" codebase="."
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-
install.html">
  <NOEMBED>
</COMMENT>
   No Java 2 support for APPLET!!
  </NOEMBED>
</EMBED>
</OBJECT>
<hr></body></html>
///:~
```

Algunas de estas líneas eran demasiado largas por lo que fue necesario envolverlas para que encajaran en la página. El código del código fuente de este libro (que se encuentra en el CD ROM de este libro, y se puede descargar de *http://www.BruceEckel.com*) funcionará sin que haya que preocuparse de corregir estos envoltorios de líneas.

El valor **code** da el nombre del archivo **.class** en el que reside el *applet*. Los valores **width** y **height** especifican el tamaño inicial del *applet* (en píxeles, como antiguamente). Hay otros elementos que se pueden ubicar dentro de la etiqueta *applet*: un lugar en el que encontrar otros archivos **.class** en Internet (**codebase**), información de alineación (**align**), un identificador especial que posibilita la intercomunicación de los *applets* (**name**), y parámetros de *applet* que proporcionan información sobre lo que ese *applet* puede recuperar. Los parámetros son de la forma:

```
<param name="identificador" value = "información">
```

y puede haber tantos como uno desee.

El paquete de código fuente de este libro proporciona una página HTML por cada *applet* de este libro, y por consiguiente muchos ejemplos de la etiqueta *applet*. Se pueden encontrar descripciones completas de los detalles de ubicación de los *applets* en las páginas web de *http://java.sun.com*.

Utilizar *Appletviewer*

El JDK de Sun (descargable gratuitamente de *http://www.sun.com*) contiene una herramienta denominada el *Appletviewer* que extrae las etiquetas <**applet**> del archivo HTML y ejecuta los *applets* sin

mostrar el texto HTML que les rodea. Debido a que el *Appletviewer* ignora todo menos las etiquetas APPLET, se puede poner esas etiquetas en el archivo fuente Java como comentarios:

```
// <applet code=MiApplet width=200 height=100>
// </applet>
```

De esta forma, se puede ejecutar "**appletviewer MiApplet.java**" y no hay que crear los pequeños archivos HTML para ejecutar pruebas. Por ejemplo, se pueden añadir las etiquetas HTML comentadas a **Applet1.java**:

```
//: c13:Applet1b.java
// Embebiendo la etiqueta applet para Appletviewer.
// <applet code=Applet1b width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;

public class Applet1b extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("¡Applet!"));
  }
} ///:~
```

Ahora, se puede invocar al *applet* con el comando:

```
appletviewer Applet1b.java
```

En este libro, se usará esta forma para probar los *applets* de manera sencilla. En breve, se verá otro enfoque de codificación que permite ejecutar *applets* desde la línea de comandos sin el *Appletviewer.*

Probar applets

Se puede llevar a cabo una prueba sencilla sin tener ninguna conexión de red, arrancando el navegador web y abriendo el archivo HTML que contiene la etiqueta *applet*. Al cargar el archivo HTML, el navegador descubrirá la etiqueta *applet* y tratará de acceder al archivo **.class** especificado por el valor **code**. Por supuesto, miremos primero la variable CLASSPATH para saber dónde buscar, y si el archivo **.class** no está en el CLASSPATH aparecerá un mensaje de error en la línea de estado del navegador indicando que no se puede encontrar ese archivo **.class**.

Cuando se desea probar esto en un sitio web, las cosas son algo más complicadas. En primer lugar, hay que *tener* un sitio web, lo que para la gran mayoría de la gente es un Proveedor de Servicios de Internet[5], un tercero, en una ubicación remota. Puesto que el *applet* es simplemente un archivo o un conjunto de archivos, el ISP no tiene que proporcionar ningún soporte especial para Java. También

[5] N. del traductor: En inglés. *Internet Service Provider* o *ISP.*

hay que tener alguna forma de mover los archivos HTML y **.class** desde un sistema al directorio correcto de la máquina ISP. Esto se suele hacer con un programa de FTP (File Transfer Protocol), de los que existen muchísimos disponibles gratuitamente o *shareware*. Por tanto, podría parecer que simplemente hay que mover los archivos a la máquina ISP con FTP, después conectarse al sitio y solicitar el archivo HTML utilizando el navegador; si aparece el *applet* y funciona, entonces todo probado, ¿verdad?

He aquí donde uno puede equivocarse. Si el navegador de la máquina cliente no puede localizar el archivo **.class** en el servidor, buscará en el CLASSPATH de la máquina *local*. Por consiguiente, podría ocurrir que no se esté cargando adecuadamente el *applet* desde el servidor, pero todo parece ir bien durante la prueba porque el navegador lo encuentra en la máquina local. Sin embargo, al conectarse otro, puede que su navegador no lo encuentre. Por tanto, al probar hay que asegurarse de borrar los archivos **.class** relevantes (o el archivo **.jar**) de la máquina local para poder verificar que existan en la ubicación adecuada dentro del servidor.

Uno de los lugares más insidiosos en los que me ocurrió esto es cuando inocentemente ubiqué un *applet* dentro de un **package**. Tras ubicar el archivo HTML y el *applet* en el servidor, resultó que el servidor no encontraba la trayectoria al *applet* debido al nombre del paquete. Sin embargo, mi navegador lo encontró en el CLASSPATH local. Por tanto, yo era el único que podía cargar el *applet* adecuadamente. Me llevó bastante tiempo descubrir que el problema era la sentencia **package**. En general, es recomendable no incorporar sentencias **package** con los *applets*.

Ejecutar *applets* desde la línea de comandos

Hay veces en las que se desearía hacer un programa con ventanas para algo más que para que éste resida en una página web. Quizás también se desearía hacer alguna de estas cosas en una aplicación "normal" a la vez que seguir disponiendo de la portabilidad instantánea de Java. En los capítulos anteriores de este libro hemos construido aplicaciones de línea de comandos, pero en algunos entornos operativos (como Macintosh, por ejemplo) no hay línea de comandos. Por tanto, por muchas razones se suele desear construir programas con ventanas pero sin *applets* haciendo uso de Java. Éste es verdaderamente un deseo muy codiciado.

La biblioteca Swing permite construir aplicaciones que preserven la apariencia del entorno operativo subyacente. Si se desea construir aplicaciones con ventanas, tiene sentido hacerlo[6] sólo si se puede hacer uso de la última versión de Java y herramientas asociadas de forma que se puedan entregar las aplicaciones sin necesidad de liar a los usuarios. Si por alguna razón uno se ve forzado a usar versiones antiguas de Java, que se lo piense bien antes de acometer la construcción de una aplicación con ventanas, especialmente, si tiene un tamaño mediano o grande.

---

[6] En mi opinión, y después de aprender Swing, no se deseará perder el tiempo con versiones anteriores.

A menudo se deseará ser capaz de crear clases que puedan invocarse bien como ventanas o bien como *applets*. Esto es especialmente apropiado cuando se prueban los *applets*, pues es generalmente mucho más rápido y sencillo ejecutar la aplicación-*applet* resultante desde la línea de comandos que arrancar el navegador web o el Appletviewer.

Para crear un *applet* que pueda ejecutarse desde la línea de comandos de la consola, simplemente hay que añadir un método **main( )** al *applet* que construya una instancia del *applet* dentro de un **JFrame**[7]. Como ejemplo sencillo, he aquí **Applet1b.java** modificado para que funcione como *applet* y como aplicación:

```
//: c13:Applet1c.java
// Un applet y una aplicación.
// <applet code=Applet1c width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1c extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("¡Applet!"));
  }
  // Un main() para la aplicación:
  public static void main(String[] args) {
    JApplet applet = new Applet1c();
    JFrame frame = new JFrame("Applet1c");
    // Para cerrar la aplicación:
    Console.setupClosing(frame);
    frame.getContentPane().add(applet);
    frame.setSize(100,50);
    applet.init();
    applet.start();
    frame.setVisible(true);
  }
} ///:~
```

Sólo se ha añadido al *applet* el método **main( )**, y el resto permanece igual. Se crea el *applet* y se añade a un **JFrame** para que pueda ser mostrado.

La línea:

```
Console.setupClosing(frame);
```

[7] Como se describió anteriormente, el AWT ya tenía el término "Frame" por lo que Swing usa el nombre JFrame.

hace que se cierre convenientemente la ventana. **Console** proviene de **com.bruceekcel.swing** y será explicada algo más tarde.

Se puede ver que en el método **main( )** se minimiza explícitamente el *applet* y se arranca, puesto que en este caso no tenemos un navegador que lo haga automáticamente. Por supuesto, así no se logra totalmente el comportamiento del navegador, que también llama a **stop( )** y **destroy( )**, pero es aceptable en la mayoría de situaciones. Si esto constituye un problema es posible forzar uno mismo estas llamadas[8].

Nótese la última línea:

```
frame.setVisible(true);
```

Sin ella, no se vería nada en pantalla.

Un marco de trabajo de visualización

Aunque el código que convierte programas en *applets* y aplicaciones a la vez produce resultados de gran valor, si se usa en todas partes, se convierte en una distracción y un derroche de papel. En vez de esto, se usará el siguiente marco de trabajo de visualización para los ejemplos Swing de todo el resto del libro:

```
//: com:bruceeckel:swing:Console.java
// Herramienta para ejecutar demos Swing desde la
// consola, tanto applets como JFrames.
package com.bruceeckel.swing;
import javax.swing.*;
import java.awt.event.*;

public class Console {
  // Crear un string título a partir del nombre de la clase:
  public static String título(Object o) {
    String t = o.getClass().toString();
    // Eliminar la palabra "class":
    if(t.indexOf("class") != -1)
      t = t.substring(6);
    return t;
  }
  public static void setupClosing(JFrame frame) {
    // La solución de JDK 1.2 como una
    // clase interna anónima:
    frame.addWindowListener(new WindowAdapter() {
```

[8] Esto tendrá sentido una vez que se haya avanzado en este capítulo. En primer lugar, se hace la referencia **JApplet** miembro **static** de la clase (en vez de una variable local del **main( )**), y después se invoca a **applet.stop( )** y **applet.destroy( )** dentro de **WindowsAdapter.windowClosing( )** antes de invocar a **System.exit( )**.

```
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      });
      // La solución mejorada del JDK 1.3:
      // frame.setDefaultCloseOperation(
      //       EXIT_ON_CLOSE);
    }
    public static void
    run(JFrame frame, int width, int height) {
      setupClosing(frame);
      frame.setSize(width, height);
      frame.setVisible(true);
    }
    public static void
    run(JApplet applet, int width, int height) {
      JFrame frame = new JFrame(título(applet));
      setupClosing(frame);
      frame.getContentPane().add(applet);
      frame.setSize(width, height);
      applet.init();
      applet.start();
      frame.setVisible(true);
    }
    public static void
    run(JPanel panel, int width, int height) {
      JFrame frame = new JFrame(title(panel));
      setupClosing(frame);
      frame.getContentPane().add(panel);
      frame.setSize(width, height);
      frame.setVisible(true);
    }
} ///:~
```

Esta herramienta puede usarse cuando se desee, por ello está en **com.bruceeckel.swing**. La clase **Console** consiste únicamente en métodos **static**. El primero se usa para extraer el nombre de la clase (usando RTTI) del objeto y para eliminar la palabra "class", que suele incorporarse en **getClass( )**. Éste usa los métodos **String indexOf( )** para determinar si está o no la palabra "class", y **substring( )** para producir la nueva cadena de caracteres sin "class" o el espacio del final. Este nombre se usa para etiquetar la ventana que mostrarán los métodos **run( )**.

El método **setupClosing( )** se usa para esconder el código que hace que un **JFrame** salga del programa al cerrarlo. El comportamiento por defecto es no hacer nada, por lo que si no se llama a **setupClosing( )** o se escribe un código equivalente para el **JFrame**, la aplicación no se cerrará. La razón por la que este código está oculto va más allá de ubicarlo directamente en los métodos **run( )** subsecuentes se debe en parte a que nos permite usar el método por sí mismo cuando lo que se desea

hacer es más complicado que lo proporcionado por **run( )**. Sin embargo, también aísla un factor de cambio: Java 2 tiene dos formas de hacer que se cierren ciertos tipos de ventanas. En el JDK 1.2, la solución es crear una nueva clase **WindowAdapter** e implementar **windowClosing( )** como se vio anteriormente (se explicará el significado completo de hacer esto algo más adelante en este capítulo). Sin embargo, durante la creación de JDK 1.3 los diseñadores de la biblioteca observaron que generalmente sería necesario cerrar las ventanas siempre que se cree algo que no sea un *applet*, por lo que añadieron el método **setDefaultCloseOperation( )** tanto a **JFrame** como a **JDialog**. Desde el punto de vista de la escritura de código, el método nuevo es mucho más sencillo de usar pero este libro se escribió mientras seguía sin implementarse JDK 1.3 en Linux y en otras plataformas, por lo que en pos de la portabilidad entre versiones, se aisló el cambio dentro de **setupClosing( )**.

El método **run( )** está sobrecargado para que funcione con **JApplets**, **JPanels** y **JFrames**. Nótese que sólo se invoca a **init( )** y **start( )** en el caso de que se trate de un **JApplet**.

Ahora, es posible ejecutar cualquier *applet* desde la consola creando un método **main( )** que contenga un método como:

```
Console.run(new MiClase(), 500, 300);
```

en la que los dos últimos argumentos son la anchura y altura de la visualización. He aquí **Applet1c.java** modificado para usar **Console**:

```
//: c13:Applet1d.java
// Console ejecuta applets desde la línea de comandos.
// <applet code=Applet1d width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1d extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("¡Applet!"));
  }
  public static void main(String[] args) {
    Console.run(new Applet1d(), 100, 50);
  }
} ///:~
```

Esto permite la eliminación de código repetido a la vez que proporciona la mayor flexibilidad posible a la hora de ejecutar los ejemplos.

Usar el Explorador de Windows

Si se usa Windows, se puede simplificar el proceso de ejecutar un programa Java de línea de comandos configurando el Explorador de Windows —el navegador de archivos de Windows, *no* el

Internet Explorer— de forma que se pueda simplemente pulsar dos veces en un **.class** para ejecutarlo. Este proceso conlleva varios pasos.

Primero, descargar e instalar el lenguaje de programación Perl de *http://www.Perl.org*. En esta misma ubicación hay documentación e instrucciones relativas al lenguaje.

A continuación, hay que crear el siguiente script sin la primera y última líneas (este script es parte del paquete de código fuente de este libro):

```
//:! C13:EjecutarJava.bat
@rem = '--*-Perl-*--
@echo off
perl -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl
@rem ';
#!perl
$file = $ARGV[0];
$file =~ s/(.*)\..*/\1/;
$file =~ s/(.*\\)*(.*)/$+/;
`java $file`;
__END__
:endofperl
///:~
```

Ahora, se abre el Explorador de Windows, se selecciona "Ver", "Opciones de Carpeta", y después se pulsa en "Tipos de Archivo". Se presiona el botón "Nuevo Tipo". En "Descripción del tipo" introducir "Fichero de clase Java". En el caso de "Extensiones Asociadas", introducir "class". Bajo "Acciones", presionar el botón "Nueva". En "Acción" introducir "Open" y en "Aplicación a utilizar para realizar la acción" introducir una línea como:

```
"c:\aaa\Perl\RunJava.bat" "%L"
```

Hay que personalizar la trayectoria ubicada antes de "EjecutarJava.bat" para que contenga la localización en la que cada uno ubique el archivo **.bat**.

Una vez hecha esta instalación, se puede ejecutar cualquier programa Java simplemente pulsando dos veces en el archivo **.class** que contenga un método **main( )**.

Hacer un botón

Hacer un botón es bastante simple: simplemente se invoca al constructor **JButton** con la etiqueta que se desee para el botón. Se verá más adelante que se pueden hacer cosas más elegantes, como poner imágenes gráficas en los botones.

Generalmente se deseará crear un campo para el botón dentro de la clase, de forma que podamos referirnos al mismo más adelante.

El **JButton** es un componente —su propia pequeña ventana— que se repintará automáticamente como parte de la actualización. Esto significa que no se pinta explícitamente ningún botón, ni ningún otro tipo de control; simplemente se ubican en el formulario y se deja que se encarguen ellos mismos de pintarse. Por tanto, para ubicar un botón en un formulario, se hace dentro de **init( )**:

```
//: c13:Boton1.java
// Poniendo botones en un applet.
// <applet code=Boton1 width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Boton1 extends JApplet {
  JButton
    b1 = new JButton("Boton 1"),
    b2 = new JButton("Boton 2");
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
  }
  public static void main(String[] args) {
    Console.run(new Button1(), 200, 50);
  }
} ///:~
```

En este caso, se ha añadido algo nuevo: antes de ubicar los elementos en el panel de contenidos, se ha asignado a éste un "gestor de disposición", de tipo **FlowLayout**. Un gestor de disposiciones permite organizar el control dentro de un formulario. El comportamiento normal para un *applet* es usar el **BorderLayout**, pero éste no funcionaría aquí pues (se aprenderá algo más adelante cuando se explique en detalle cómo controlar la disposición de un formulario) por defecto cubre cada control completamente con cada uno que se añada. Sin embargo, **FlowLayout** hace que los controles fluyan por el formulario, de izquierda a derecha y de arriba hacia abajo.

Capturar un evento

Uno se dará cuenta de que si se compila y ejecuta el *applet* de arriba, no ocurre nada cuando se presionan los botones. Es aquí donde hay que escribir algún tipo de código para ver qué ocurrirá. La base de la programación conducida por eventos, que incluye mucho de lo relacionado con IGU, es atar los eventos al código que responda a los mismos.

La forma de hacer esto en Swing es separando limpiamente la interfaz (los componentes gráficos) y la implementación (el código que se desea ejecutar cuando se da cierto evento en un componen-

te). Cada componente Swing puede reportar todos los eventos que le pudieran ocurrir, y puede reportar también cada tipo de evento individualmente. Por tanto, si uno no está interesado, por ejemplo, en ver si se está moviendo el ratón por encima del botón, no hay por qué registrar interés en ese evento. Se trata de una forma muy directa y elegante de manejar la programación dirigida por eventos, y una vez que se entienden los conceptos básicos se puede usar componentes Swing de forma tan sencilla que nunca antes se podría imaginar —de hecho, el modelo se extiende a todo lo que pueda clasificarse como JavaBean (sobre los que se aprenderá más adelante en este Capítulo).

En primer lugar, nos centraremos simplemente en el evento de interés principal para los componentes que se usen. En el caso de un **JButton**, este "evento de interés" es que se presione el botón. Para registrar interés en que se presione un botón, se invoca al método **addActionListener( )** del **JButton**. Este método espera un parámetro que es un objeto que implemente la interfaz **ActionListener**, que contiene un único método denominado **actionPerformed( )**. Por tanto, todo lo que hay que hacer para adjuntar código a un **JButton** es implementar la interfaz **ActionListener** en una clase y registrar un evento de esa clase con el **JButton** vía **addActionListener( )**. El método será invocado al presionar el botón (a esto se le suele denominar *retrollamada*).

Pero, ¿cuál debería ser el resultado de presionar ese botón? Nos gustaría ver que algo cambia en la pantalla, por tanto, se introducirá un nuevo componente Swing: el **JTextField**. Se trata de un lugar en el que se puede escribir texto, o en este caso, ser modificado por el programa. Aunque hay varias formas de crear un **JTextField**, la más sencilla es decir al constructor lo ancho que se desea que sea éste. Una vez ubicado el **JTextField** en el formulario, se puede modificar su contenido utilizando el método **setText( )** (hay muchos otros métodos en **JTextField**, que pueden encontrarse en la documentación HTML del JDK disponible en *http://java.sun.com*). Ésta es la apariencia que tiene:

```
//: c13:Boton2.java
// Respondiendo al presionar un botón.
// <applet code=Boton2 width=200 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Boton2 extends JApplet {
  JButton
    b1 = new JButton("Boton 1"),
    b2 = new JButton("Boton 2");
  JTextField txt = new JTextField(10);
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e){
      String nombre =
        ((JButton)e.getSource()).getText();
      txt.setText(nombre);
    }
  }
```

```
  BL al = new BL();
  public void init() {
    b1.addActionListener(al);
    b2.addActionListener(al);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(txt);
  }
  public static void main(String[] args) {
    Console.run(new Button2(), 200, 75);
  }
} ///:~
```

Crear un **JTextFiled** y ubicarlo en el lienzo conlleva los mismos pasos que en los **JButtons**, o cualquier componente Swing. La diferencia en el programa de arriba radica en la creación de **BL** de la clase **ActionLitener** ya mencionada. El argumento al **actionPerformed( )** es de tipo **ActionEvent**, que contiene toda la información sobre el evento y su procedencia. En este caso, quería describir el botón que se presionaba: **getSource( )** produce el objeto en el que se originó el evento, y asumí que **JButton.getText( )** devuelve el texto que está en el botón, y que éste está en el **JTextField** para probar que sí que se estaba invocando al código al presionar el botón.

En **init( )** se usa **addActionListener( )** para registrar el objeto **BL** con ambos botones.

Suele ser más conveniente codificar el **ActionListener** como una clase interna anónima, especialmente desde que se tiende sólo a usar una instancia de cada clase oyente. Puede modificarse **Boton2.java** para usar clases internas anónimas como sigue:

```
//: c13:Boton2b.java
// Utilizando clases anónimas internas.
// <applet code=Boton2b width=200 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Boton2b extends JApplet {
  JButton
    b1 = new JButton("Boton 1"),
    b2 = new JButton("Boton 2");
  JTextField txt = new JTextField(10);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String nombre =
```

```
            ((JButton)e.getSource()).getText();
         txt.setText(nombre);
      }
   };
   public void init() {
      b1.addActionListener(al);
      b2.addActionListener(al);
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(b1);
      cp.add(b2);
      cp.add(txt);
   }
   public static void main(String[] args) {
      Console.run(new Button2b(), 200, 75);
   }
} ///:~
```

Se preferirá el enfoque de usar clases anónimas internas (siempre que sea posible) para los ejemplos de este libro.

Áreas de texto

Un **JTextArea** es como un **JTextField** excepto en que puede tener múltiples líneas y tiene más funcionalidad. Un método particularmente útil es **append( )**; con él se puede incorporar alguna salida de manera sencilla a la **JTextArea**, logrando así una mejora de Swing (dado que se puede hacer desplazamiento hacia delante y hacia atrás) sobre lo que se había logrado hasta la fecha haciendo uso de programas de línea de comandos que imprimían a la salida estándar. Como ejemplo, el siguiente programa rellena una **JTextArea** con la salida del generador **geografía** del Capítulo 9:

```
//: c13:AreaTexto.java
// Utilizando el control JTextArea.
// <applet code=TextArea width=475 height=425>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class AreaTexto extends JApplet {
   JButton
      b = new JButton("Añadir datos"),
      c = new JButton("Limpiar datos");
```

```
  JTextArea t = new JTextArea(20, 40);
  Map m = new HashMap();
  public void init() {
    // Usar todos los datos:
    Colecciones2.fill(m,
      Colecciones2.geografia,
      CapitalesPaises.pares.length);
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        for(Iterator it= m.entrySet().iterator();
            it.hasNext();){
          Map.Entry me = (Map.Entry)(it.next());
          t.append(me.getKey() + ": "
            + me.getValue() + "\n");
        }
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText("");
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(b);
    cp.add(c);
  }
  public static void main(String[] args) {
    Console.run(new TextArea(), 475, 425);
  }
} ///:~
```

En el método **init( )**, se rellena **Map** con todos los países y sus capitales. Nótese que para ambos botones se crea y añade el **ActionListener** sin definir una variable intermedia, puesto que nunca es necesario hacer referencia a ese oyente en todo el programa. El botón "Añadir Datos" da formato e incorpora todos los datos, mientras que el botón "Borrar Datos" hace uso de **setText( )** para eliminar todo el texto del **JTextArea**.

Al añadir el **JTextArea** al *applet*, se envuelve en un **JScrollPane** para controlar el desplazamiento cuando se coloca demasiado texto en la pantalla. Esto es todo lo que hay que hacer para lograr capacidades de desplazamiento al completo. Habiendo intentado averiguar cómo hacer el equivalente en otros entornos de programación, estoy muy impresionado con la simplicidad y el buen diseño de componentes como **JScrollPane**.

Controlar la disposición

La forma de colocar los componentes en un formulario en Java probablemente sea distinta de cualquier otro sistema IGU que se haya usado. En primer lugar, es todo código; no hay "recursos" que controlen la ubicación de los componentes. Segundo, la forma de colocar los componentes en un formulario no está controlada por ningún tipo de posicionado absoluto sino por un "gestor de disposición" que decide cómo disponer los componentes basándose en el orden en que se invoca a **add( )** para ellos. El tamaño, forma y ubicación de los componentes será notoriamente diferente de un gestor de disposición a otro. Además, los gestores de disposición se adaptan a las dimensiones del *applet* o a la ventana de la aplicación, por lo que si se cambia la dimensión de la ventana, cambiarán como respuesta el tamaño, la forma y la ubicación de los componentes.

JApplet, JFrame, JWindow y **JDialog** pueden todos producir un **Container** con **getContentPane( )** que puede contener y mostrar **Componentes**. En **Container**, hay un método denominado **setLayout( )** que permite elegir entre varios gestores de disposición distintos. Otras clases, como **JPanel**, contienen y muestran los componentes directamente, de forma que también se establece el gestor de disposición directamente, sin usar el cuadro de contenidos.

En esta sección se explorarán los distintos gestores de disposición ubicando botones en los mismos (puesto que es lo más sencillo que se puede hacer con ellos). No habrá ninguna captura de eventos de los botones puesto que simplemente se pretende que estos ejemplos muestren cómo se disponen los botones.

BorderLayout

El *applet* usa un esquema de disposición por defecto: el **BorderLayout** (varios de los ejemplos anteriores variaban éste al **FlowLayout**). Sin otra instrucción, éste toma lo que se **add( )** (añade) al mismo y lo coloca en el centro, extendiendo el objeto hasta los bordes.

Sin embargo, hay más. Este gestor de disposición incorpora los conceptos de cuatro regiones limítrofes en torno a un área central. Cuando se añade algo a un panel que está haciendo uso de un **BorderLayout** se puede usar el método sobrecargado **add( )** que toma como primer parámetro un valor constante. Este valor puede ser cualquiera de los siguientes:

BorderLayout.NORTH (superior)
BorderLayout.SOUTH (inferior)
BorderLayout.EAST (derecho)
BorderLayout.WEST (izquierdo)
BorderLayout.CENTER (rellenar el centro hasta los bordes de todos los demás componentes)

Si no se especifica un área al ubicar un objeto, éste ira por defecto a **CENTER**.

He aquí un simple ejemplo. Se usa la disposición por defecto, puesto que **JApplet** se pone por defecto a **BorderLayout**:

```
//: c13:BorderLayout1.java
// Demuestra BorderLayout.
// <applet code=BorderLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BorderLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.add(BorderLayout.NORTH,
       new JButton ("Norte"));
    cp.add(BorderLayout.SOUTH,
       new JButton ("Sur"));
    cp.add(BorderLayout.EAST,
       new JButton ("Este"));
    cp.add(BorderLayout.WEST,
       new JButton ("Oeste"));
    cp.add(BorderLayout.CENTER,
       new JButton ("Centro"));
  }
  public static void main(String[] args) {
    Console.run(new BorderLayout1(), 300, 250);
  }
} ///:~
```

Para todas las ubicaciones excepto **CENTER**, el elemento que se añada se comprime hasta caber en la menor cantidad de espacio posible en una de las dimensiones, extendiéndose al máximo en la otra dimensión. Sin embargo, **CENTER** se extiende en ambas dimensiones para ocupar todo el centro.

FlowLayout

Simplemente "hace fluir" los componentes del formulario de izquierda a derecha hasta que se llena el espacio de arriba, después se mueve una fila hacia abajo y continúa fluyendo.

He aquí un ejemplo que establece el gestor de disposición a **FlowLayout** y después ubica botones en el formulario. Se verá que con **FlowLayout** los componentes tomarán su tamaño "natural". Por ejemplo un **JButton**, será del tamaño de su cadena de caracteres:

```
//: c13:FlowLayout1.java
// Demuestra FlowLayout.
// <applet code=FlowLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
public class FlowLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < 20; i++)
      cp.add(new JButton("Boton " + i));
  }
  public static void main(String[] args) {
    Console.run(new FlowLayout1(), 300, 250);
  }
} ///:~
```

Se compactarán todos los componentes a su menor tamaño posible en un **FlowLayout**, de forma que se podría obtener un comportamiento ligeramente sorprendente. Por ejemplo, dado que un **JLabel** será del tamaño de su cadena de caracteres, intentar justificar el texto a la derecha conduce a que lo mostrado utilizando **FlowLayout** no varíe.

GridLayout

Un **GridLayout** permite construir una tabla de componentes, y al añadirlos se ubican de izquierda a derecha y de arriba abajo en la rejilla. En el constructor se especifica el número de filas y columnas que se necesiten y éstas se distribuyen en proporciones iguales:

```
//: c13:GridLayout1.java
// Demuestra GridLayout.
// <applet code=GridLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class GridLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(7,3));
    for(int i = 0; i < 20; i++)
      cp.add(new JButton("Boton " + i));
  }
  public static void main(String[] args) {
    Console.run(new GridLayout1(), 300, 250);
  }
} ///:~
```

En este caso hay 21 casillas y sólo 20 botones. La última casilla se deja vacía porque en un **GridLayout** no se produce ningún "balanceo".

GridBagLayout

El **GridBagLayout** proporciona un control tremendo a la hora de decidir exactamente cómo distribuir en sí las regiones de la ventana además de cómo reformatear cada región cuando se redimensionan las ventanas. Sin embargo, también es el gestor de disposiciones más complicado, y bastante difícil de entender. Inicialmente está pensado para generación automática de código por parte de un constructor de IGU (los buenos constructores de IGU utilizarán **GridBagLayout** en vez de posicionamiento absoluto). Si uno tiene un diseño tan complicado que cree que debe usar **GridBagLayout**, habría que usar una buena herramienta de construcción de IGU para ese diseño. Si siente la necesidad de saber detalles intrínsecos, le recomiendo la lectura de *Core Java 2*, de Hostmann & Cornell (Prentice-Hall, 1999), o un libro dedicado a la Swing, como punto de partida.

Posicionamiento absoluto

También es posible establecer la posición absoluta de los componentes gráficos así:

1. Poner a **null** el gestor de disposiciones del **Container**: **setLayout(null)**.

2. Invocar a **setBounds( )** o **reshape( )** (en función de la versión del lenguaje) por cada componente, pasando un rectángulo circundante con sus coordenadas en puntos. Esto se puede hacer en el constructor o en **paint( )**, en función de lo que se desee lograr.

Algunos constructores de IGU usan este enfoque de forma extensiva, pero ésta no suele ser la mejor forma de generar código. Los constructores de IGU más útiles usan en vez de éste el **GridBagLayout**.

BoxLayout

Debido a que la gente tiene tantas dificultades a la hora de entender y trabajar con **GridBagLayout**, Swing también incluye el **BoxLayout**, que proporciona muchos de los beneficios de **GridBagLayout** sin la complejidad, de forma que a menudo se puede usar cuando se precisen disposiciones codificadas a mano (de nuevo, si el diseño se vuelve demasiado complicado, es mejor usar un constructor de IGU que genere **GridBagLayouts** automáticamente). **BoxLayout** permite controlar la ubicación de los componentes vertical u horizontalmente, y controlar el espacio entre componentes utilizando algo que denomina "puntales y pegamento". En primer lugar, veremos cómo usar directamente **BoxLayout** de la misma forma que se ha demostrado el uso de los demás gestores de disposiciones:

```
//: c13:BoxLayout1.java
// BoxLayouts vertical y horizontal.
// <applet code=BoxLayout1
// width=450 height=200> </applet>
import javax.swing.*;
```

```
import java.awt.*;
import com.bruceeckel.swing.*;

public class BoxLayout1 extends JApplet {
  public void init() {
    JPanel jpv = new JPanel();
    jpv.setLayout(
      new BoxLayout(jpv, BoxLayout.Y_AXIS));
    for(int i = 0; i < 5; i++)
      jpv.add(new JButton("" + i));
    JPanel jph = new JPanel();
    jph.setLayout(
      new BoxLayout(jph, BoxLayout.X_AXIS));
    for(int i = 0; i < 5; i++)
      jph.add(new JButton("" + i));
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, jpv);
    cp.add(BorderLayout.SOUTH, jph);
  }
  public static void main(String[] args) {
    Console.run(new BoxLayout1(), 450, 200);
  }
} ///:~
```

El constructor de **BoxLayout** es un poco diferente del de los otros gestores de disposición —se proporciona el **Container** que va a ser controlado por el **BoxLayout** como primer argumento, y la dirección de la disposición como segundo argumento.

Para simplificar las cosas, hay un contenedor especial denominado **Box** que usa **BoxLayout** como gestor nativo. El ejemplo siguiente distribuye los componentes horizontal y verticalmente usando **Box**, que tiene dos métodos **static** para crear cajas con alineación vertical y horizontal:

```
//: c13:Box1.java
// BoxLayout vertical y horizontal.
// <applet code=Box1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box1 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    for(int i = 0; i < 5; i++)
      bv.add(new JButton("" + i));
```

```
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++)
      bh.add(new JButton("" + i));
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, bv);
    cp.add(BorderLayout.SOUTH, bh);
  }
  public static void main(String[] args) {
    Console.run(new Box1(), 450, 200);
  }
} ///:~
```

Una vez que se tiene una **Box**, se pasa como segundo parámetro al añadir componentes al panel contenedor.

Los puntales añaden espacio entre componentes, midiéndose este espacio en puntos. Para usar un puntal, simplemente se añade éste entre la adición de los componentes que se desea separar:

```
//: c13:Box2.java
// Añadiendo puntales.
// <applet code=Box2
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box2 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    for(int i = 0; i < 5; i++) {
      bv.add(new JButton("" + i));
      bv.add(Box.createVerticalStrut(i*10));
    }
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++) {
      bh.add(new JButton("" + i));
      bh.add(Box.createHorizontalStrut(i*10));
    }
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, bv);
    cp.add(BorderLayout.SOUTH, bh);
  }
  public static void main(String[] args) {
    Console.run(new Box2(), 450, 300);
  }
} ///:~
```

Los puntales separan los componentes en una cantidad fija, mientras que la cola actúa al contrario: separa los componentes tanto como sea posible. Por consiguiente, es más un elástico que "pegamento" (y el diseño en el que se basó se denominaba "elásticos y puntales" por lo que la elección del término es algo misteriosa).

```
//: c13:Box3.java
// Usando Pegamento.
// <applet code=Box3
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box3 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    bv.add(new JLabel("Hola"));
    bv.add(Box.createVerticalGlue());
    bv.add(new JLabel("Applet"));
    bv.add(Box.createVerticalGlue());
    bv.add(new JLabel("Mundo"));
    Box bh = Box.createHorizontalBox();
    bh.add(new JLabel("Hola"));
    bh.add(Box.createHorizontalGlue());
    bh.add(new JLabel("Applet"));
    bh.add(Box.createHorizontalGlue());
    bh.add(new JLabel("Mundo"));
    bv.add(Box.createVerticalGlue());
    bv.add(bh);
    bv.add(Box.createVerticalGlue());
    getContentPane().add(bv);
  }
  public static void main(String[] args) {
    Console.run(new Box3(), 450, 300);
  }
} ///:~
```

Un puntal funciona de forma inversa, pero un área rígida fija los espacios entre componentes en ambas direcciones:

```
//: c13:Box4.java
// Las áreas rígidas son como pares de puntales.
// <applet code=Box4
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
```

```
import com.bruceeckel.swing.*;

public class Box4 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    bv.add(new JButton("Arriba"));
    bv.add(Box.createRigidArea(
      new Dimension(120, 90)));
    bv.add(new JButton("Abajo"));
    Box bh = Box.createHorizontalBox();
    bh.add(new JButton("Izquierda"));
    bh.add(Box.createRigidArea(
      new Dimension(160, 80)));
    bh.add(new JButton("Derecha"));
    bv.add(bh);
    getContentPane().add(bv);
  }
  public static void main(String[] args) {
    Console.run(new Box4(), 450, 300);
  }
} ///:~
```

Habría que ser consciente de que las áreas rígidas son algo controvertidas. Puesto que usan valores absolutos, algunos piensan que pueden causar demasiados problemas como para merecer la pena.

¿El mejor enfoque?

Swing es potente; puede lograr mucho con unas pocas líneas de código. Los ejemplos mostrados en este libro son razonablemente simples, y por propósitos de aprendizaje tiene sentido escribirlos a mano. De hecho se puede lograr, simplemente, combinando disposiciones. En algún momento, sin embargo, deja de tener sentido codificar a mano formularios IGU —se vuelve demasiado complicado y no constituye un buen uso del tiempo de programación. Los diseñadores de Java y Swing orientaron el lenguaje y las bibliotecas de forma que soportaran herramientas de construcción de IGU, creadas con el propósito específico de facilitar la experiencia de programación. A medida que se entiende todo lo relacionado con disposiciones y con cómo tratar con esos eventos (como se describe a continuación), no es particularmente importante que se sepan los detalles de cómo distribuir componentes a mano —deje que la herramienta apropiada haga el trabajo por usted (Java, después de todo, está diseñado para incrementar la productividad del programador).

El modelo de eventos de Swing

En el modelo de eventos de Swing, un componente puede iniciar ("disparar") un evento. Cada tipo de evento está representado por una clase distinta. Cuando se dispara un evento, éste es recibido

por uno o más "oyentes", que actúan sobre ese evento. Por consiguiente, la fuente de un evento y el lugar en el que éste es gestionado podrían ser diferentes. Por tanto, generalmente se usan los componentes Swing tal y como son, aunque es necesario escribir el código al que se invoca cuando los componentes reciben un evento, de forma que nos encontramos ante un excelente ejemplo de la separación entre la interfaz y la implementación.

Cada oyente de eventos es un objeto de una clase que implementa un tipo particular de **interfaz** oyente. Por tanto, como programador, todo lo que hay que hacer es crear un objeto oyente y registrarlo junto con el componente que dispara el evento. El registro se lleva a cabo invocando a un método **addXXXListener( )** en el componente que dispara el evento, donde "XXX" representa el tipo de evento por el que se escucha. Se pueden conocer fácilmente los tipos de eventos que pueden gestionarse fijándose en los nombres de los métodos "addListener", y si se intenta escuchar por eventos erróneos se descubrirá el error en tiempo de compilación. Más adelante en este capítulo se verá que los JavaBeans también usan los nombres de los métodos "addListener" para determinar qué eventos puede manejar un Bean.

Después, toda la lógica de eventos irá dentro de una clase oyente. Cuando se crea una clase oyente, su única restricción es que debe implementar la interfaz apropiada. Se puede crear una clase oyente global, pero ésta es una situación en la que tienden a ser bastante útiles las clases internas, no sólo por proporcionar una agrupación lógica de las clases oyentes dentro del IU o de las clases de la lógica de negocio a las que sirven, sino (como se verá más adelante) por el hecho de que una clase interna proporciona una forma elegante más allá de una clase y los límites del subsistema.

Todos los ejemplos mostrados en este capítulo hasta el momento han usado el modelo de eventos de Swing, pero el resto de la sección acabará de describir los detalles de ese modelo.

Tipos de eventos y oyentes

Todos los componentes Swing incluyen **addXXXListener( )** y **removeXXXListener( )** de forma que se pueden añadir y eliminar los tipos de oyentes apropiados de cada componente. Se verá que en cada caso el "XXX" representa el parámetro del método, por ejemplo: **addMiOyente (MiOyente m)**. La tabla siguiente incluye los eventos básicos asociados, los oyentes y los métodos, junto con los componentes básicos que soportan esos eventos particulares proporcionando los métodos **addXXXListener( )** y **removeXXXListener( )**. Debería tenerse en mente que el modelo de eventos fue diseñado para ser extensible, de forma que se podrían encontrar otros tipos de eventos y oyentes que no estén en esta tabla.

Evento, interfaz oyente y métodos add- y remove	Componentes que soportan este evento
ActionEvent **ActionListener** **addActionListener( )** **removeActionListener( )**	**JButton**, **JList**, **JTextField**, **JMenuItem** y sus derivados incluyendo **JCheckBoxMenuItem**, **JMenu**, y **JPopupMenu**.

Evento, interfaz oyente y métodos add- y remove	Componentes que soportan este evento
AdjustmentEvent **AdjustmentListener** **addAdjustementListener( )** **removeAdjustmentListener( )**	**JScrollBar** y cualquier cosa que se cree que implemente la **interfaz Adjustable**.
ComponentEvent **ComponentListener** **addComponentListener( )** **removeComponentListener( )**	***Component** y sus derivados incluyendo **JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea** y **JTextField**.
ContainerEvent **ContainerListener** **addContainerListener( )** **removeContainerListener( )**	**Container** y sus derivados incluyendo **JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog** y **JFrame**.
FocusEvent **FocusListener( )** **addFocusListener( )** **removeFocusListener( )**	**Component** y sus derivados*.
KeyListener **addKeyListener ( )** **removeKeyListener( )**	**Component** y sus derivados*.
MouseEvent (para ambos clicks y movimiento) **MouseListener** **addMouseListener( )** **removeMouseListener( )**	**Component** y sus derivados*.
MouseEvent[9] (para ambos clics y movimiento) **MouseMotionListener** **addMouseMotionListener( )** **removeMouseMotionLitener( )**	**Component** y sus derivados*.
WindowEvent **WindowListener** **addWindowListener( )** **removeWindowListener( )**	**Window** y sus derivados, incluyendo **JDialog, JFileDialog** y **JFrame**.

[9] No existe evento **MouseMotionEvent** incluso aunque parece que debería haberlo. Hacer clic y el movimiento se combinan en **MouseEvent**, por lo que esta segunda ocurrencia de **MouseEvent** en la tabla no es ningún error.

Evento, interfaz oyente y métodos add- y remove	Componentes que soportan este evento
ItemEvent **ItemListener** **addItemListener( )** **removeItemListener( )**	**JCheckBox,** **JCheckBoxMenuItem,** **JComboBox, JList** y cualquier cosa que implemente la **interfaz ItemSelectable.**
TextEvent **TextListener** **addTextListener( )** **removeTextListener( )**	Cualquier cosa derivada de **JTextComponent** incluyendo **JTextArea** y **JTextField**.

Se puede ver que cada tipo de componente sólo soporta ciertos tipos de eventos. Se vuelve bastante difícil mirar todos los eventos que soporta cada componente. Un enfoque más sencillo es modificar el programa **MostrarLimpiarMetodos.java** del Capítulo 12, de forma que muestre todos los oyentes de eventos soportados por cualquier componente Swing que se introduzca.

El Capítulo 12 introdujo la *reflectividad* y usaba esa faceta para buscar los métodos de una clase particular —bien la lista de métodos al completo, o bien un subconjunto de aquéllos cuyos nombres coincidan con la palabra clave que se proporcione. La magia de esto es que puede mostrar automáticamente *todos* los métodos de una clase sin forzarla a recorrer la jerarquía de herencias hacia arriba examinando las clases base en cada nivel. Por consiguiente, proporciona una valiosa herramienta para ahorrar tiempo a la hora de programar: dado que la mayoría de los nombres de los métodos de Java son bastante descriptivos, se pueden buscar los nombres de métodos que contengan una palabra de interés en particular. Cuando se encuentre lo que uno cree estar buscando, compruebe la documentación en línea.

Sin embargo, cuando llegamos al Capítulo 12 no se había visto Swing, por lo que la herramienta de ese capítulo se desarrolló como aplicación de línea de comandos. Aquí está la versión más útil de IGU, especializada en buscar los métodos "addListener" de los componentes Swing:

```
//: c13:MostrarAddListeners.java
// Mostrar los metodos "addXXXListener" de cualquier
// clase Swing.
// <applet code = MostrarAddListeners
// width=500 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.io.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;
```

```
public class MostrarAddListeners extends JApplet {
  Class cl;
  Method[] m;
  Constructor[] ctor;
  String[] n = new String[0];
  JTextField nombre = new JTextField(25);
  JTextArea resultados = new JTextArea(40, 65);
  class NombreL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nm = nombre.getText().trim();
      if(nm.length() == 0) {
        resultados.setText("No coinciden");
        n = new String[0];
        return;
      }
      try {
        cl = Class.forName("javax.swing." + nm);
      } catch(ClassNotFoundException ex) {
        resultados.setText("No coinciden");
        return;
      }
      m = cl.getMethods();
      // Convertir en un array de Strings:
      n = new String[m.length];
      for(int i = 0; i < m.length; i++)
        n[i] = m[i].toString();
      representar();
    }
  }
  void representar() {
    // Crear el conjunto resultado:
    String[] rs = new String[n.length];
    int j = 0;
    for (int i = 0; i < n.length; i++)
      if(n[i].indexOf("add") != -1 &&
        n[i].indexOf("Listener") != -1)
          rs[j++] =
            n[i].substring(n[i].indexOf("add"));
    resultados.setText("");
    for (int i = 0; i < j; i++)
      resultados.append(
        EliminarCalificadores.strip(rs[i]) + "\n");
  }
  public void init() {
    nombre.addActionListener(new NombreL());
```

```
      JPanel cima = new JPanel();
      cima.add(new JLabel(
        "Nombre de clase Swing (presionar ENTER):"));
      cima.add(nombre);
      Container cp = getContentPane();
      cp.add(BorderLayout.NORTH, top);
      cp.add(new JScrollPane(resultados));
   }
   public static void main(String[] args) {
      Console.run(new MostrarAddListeners(), 500,400);
   }
} ///:~
```

Aquí se vuelve a usar la clase **EliminarCalificadores** definida en el Capítulo 12 importando la biblioteca **com.bruceeckel.util**.

La IGU contiene un **JTextField nombre** en el que se puede introducir el nombre de la clase Swing que se desea buscar. Los resultados se muestran en una **JTextArea**.

Se verá que no hay botones en los demás componentes mediante los que indicar que se desea comenzar la búsqueda. Esto se debe a que **JTextField** está monitorizada por un **ActionListener**. Siempre que se haga un cambio y se presione ENTER se actualiza la lista inmediatamente. Si el texto no está vacío, se usa dentro de **Class.forName( )** para intentar buscar la clase. Si el nombre es incorrecto, **Class.forName( )** fallará, lo que significa que lanzará una excepción. Ésta se atrapa y se pone la **JTextArea** a "No coinciden". Pero si se teclea un nombre correcto (teniendo en cuenta las mayúsculas y minúsculas), **Class.forName( )** tiene éxito y **getMethods( )** devolverá un array de objetos **Method**. Cada uno de los objetos del array se convierte en un **String** vía **toString( )** (esto produce la signatura completa del método) y se añade a **n**, un array de **Strings**. El array **n** es un miembro de **class ShowAddListeners** y se usa en la actualización de la pantalla siempre que se invoca a **representar( )**.

El método **representar( )** crea un array de **Strings** llamado **rs** (de "result set" —"conjunto resultado" en inglés). De este conjunto se copian en **n** aquellos elementos que contienen "add" y "Listener". Después se usan **indexOf( )** y **substring( )** para eliminar los calificadores como **public**, **static**, etc. Finalmente, **EliminarCalificadores.strip( )** remueve los calificadores de nombre extra.

Este programa constituye una forma conveniente de investigar las capacidades de un componente Swing. Una vez que se conocen los eventos soportados por un componente en particular, no es necesario buscar nada para que reaccione ante el evento. Simplemente:

1. Se toma el nombre de la clase evento y se retira la palabra "**Event**". Se añade la palabra "**Listener**" a lo que queda. Ésta es la interfaz oyente a implementar en la clase interna.

2. Implementar la interfaz de arriba y escribir los métodos de los eventos a capturar. Por ejemplo, se podría estar buscando movimientos de ratón, por lo que se escribe código para el método **mouseMoved( )** de la interfaz **MouseMotionListener**. (Hay que implementar los otros métodos, por supuesto, pero a menudo hay un atajo que veremos pronto.)

3. Crear un objeto de la clase oyente del paso 2. Registrarlo con el componente con el método producido al prefijar "**add**" al nombre del Oyente. Por ejemplo, **addMouseMotionListener( )**.

He aquí algunos de las interfaces Oyente:

Listener interface oyente y adaptadores	Métodos de la interfaz
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Éste no es un listado exhaustivo, en parte porque el modelo de eventos permite crear tipos de eventos y oyentes personalizados. Por consiguiente, frecuentemente se tendrá acceso a bibliotecas que han inventado sus propios eventos, de forma que el conocimiento que se adquiera en este capítulo te permitirá adivinar como utilizar esos eventos.

Utilizar adaptadores de oyentes por simplicidad

En la tabla de arriba, se puede ver que algunas interfaces oyentes sólo tienen un método. Éstas son triviales de implementar puesto que se implementarán sólo cuando se desee escribir ese método en particular. Sin embargo, las interfaces oyentes que tienen múltiples métodos pueden ser menos agradables de usar. Por ejemplo, algo que hay que hacer siempre que se cree una aplicación es proporcionar un **WindowListener** al **JFrame** de forma que cuando se logre el evento **windowClosing( )** se llame a **System.exit( )** para salir de la aplicación. Pero dado que **WindowListener** es una **interfaz**, hay que implementar todos los demás métodos incluso aunque no hagan nada. Esto puede resultar molesto.

Para solucionar el problema, algunas (aunque no todas) de las interfaces oyentes que tienen más de un método se suministran con *adaptadores*, cuyos nombres pueden verse en la tabla de arriba. Cada adaptador proporciona métodos por defecto vacíos para cada uno de los métodos de la interfaz. Después, todo lo que hay que hacer es heredar del adaptador y superponer sólo los métodos que se necesiten cambiar. Por ejemplo, el **WindowListener** típico a usar tendrá la siguiente apariencia (recuérdese que se ha envuelto en la clase **Console** de **com.bruceeckel.swing**):

```
class MiWindowListener extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

La única razón para la existencia de los adaptadores es facilitar la creación de las clases Oyente.

Sin embargo, los adaptadores también tienen un inconveniente. Imagínese que se escribe un **WindowAdapter** como el de arriba:

```
class MiWindowListener extends WindowAdapter {
  public void WindowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

No funciona, pero uno se volverá loco tratando de averiguar por qué, pues compila y se ejecuta correctamente —excepto por el hecho de que cerrar la ventana no saldrá del programa. ¿Se ve el problema? Está en el nombre del método: **WindowClosing( )** en vez de **windowClosing( )**. Un simple error con una mayúscula produce la adición de un método completamente nuevo. Sin embargo, no es el método que se invoca cuando se cierra la ventana, por lo que no se obtendrán los resultados deseados. Con excepción de este inconveniente, una **interfaz** garantizará que los métodos estén correctamente implementados.

Seguimiento de múltiples eventos

Para probar que estos eventos se están disparando verdaderamente, y como experimento interesante, merece la pena crear un *applet* que haga un seguimiento de comportamiento extra en un **JButton** (que no sea si está o no presionado). Este ejemplo también muestra cómo heredar tu propio objeto botón puesto que se usa como destino de todos los eventos de interés. Para lograrlo, simplemente se puede heredar de **JButton**[10].

La clase **MiBoton** es una clase interna de **RastrearEvento**, por lo que **MiBoton** puede llegar a la ventana padre y manipular sus campos de texto, que es lo necesario para poder escribir la información de estado en los campos del padre. Por supuesto ésta es una solución limitada, puesto que **MiBoton** puede usarse sólo en conjunción con **RastrearEvento**. A este tipo de código se le suele denominar "altamente acoplado":

```
//: c13:RastrearEvento.java
// Mostrar eventos a medida que ocurren.
// <applet code=RastrearEvento
//  width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class RastrearEvento extends JApplet {
  HashMap h = new HashMap();
  String[] evento = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited","mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  MiBoton
    b1 = new MiBoton(Color.blue, "test1"),
    b2 = new MiBoton(Color.red, "test2");
  class MiBoton extends JButton {
    void informa(String campo, String msg) {
      ((JTextField)h.get(campo)).setText(msg);
    }
    FocusListener fl = new FocusListener() {
      public void focusGained(FocusEvent e) {
        informar("focusGained", e.paramString());
```

[10] En Java 1.0/1.1 *no* se podía heredar de manera útil del objeto botón. Esto no era sino uno de los muchos fallos de diseño de que adolecía.

```
      }
      public void focusLost(FocusEvent e) {
        informar("focusLost", e.paramString());
      }
    };
    KeyListener kl = new KeyListener() {
      public void keyPressed(KeyEvent e) {
        informar("keyPressed", e.paramString());
      }
      public void keyReleased(KeyEvent e) {
        informar("keyReleased", e.paramString());
      }
      public void keyTyped(KeyEvent e) {
        informar("keyTyped", e.paramString());
      }
    };
    MouseListener ml = new MouseListener() {
      public void mouseClicked(MouseEvent e) {
        informar("mouseClicked", e.paramString());
      }
      public void mouseEntered(MouseEvent e) {
        informar("mouseEntered", e.paramString());
      }
      public void mouseExited(MouseEvent e) {
        informar("mouseExited", e.paramString());
      }
      public void mousePressed(MouseEvent e) {
        informar("mousePressed", e.paramString());
      }
      public void mouseReleased(MouseEvent e) {
        informar("mouseReleased", e.paramString());
      }
    };
    MouseMotionListener mml =
      new MouseMotionListener() {
      public void mouseDragged(MouseEvent e) {
        informar("mouseDragged", e.paramString());
      }
      public void mouseMoved(MouseEvent e) {
        informar("mouseMoved", e.paramString());
      }
    };
    public MiBoton(Color color, String etiqueta) {
      super(etiqueta);
      setBackground(color);
```

```
      addFocusListener(fl);
      addKeyListener(kl);
      addMouseListener(ml);
      addMouseMotionListener(mml);
    }
  }
  public void init() {
    Container c = getContentPane();
    c.setLayout(new GridLayout(evento.length+1,2));
    for(int i = 0; i < evento.length; i++) {
      JTextField t = new JTextField();
      t.setEditable(false);
      c.add(new JLabel(event[i], JLabel.RIGHT));
      c.add(t);
      h.put(evento[i], t);
    }
    c.add(b1);
    c.add(b2);
  }
  public static void main(String[] args) {
    Console.run(new RastrearEvento(), 700, 500);
  }
} ///:~
```

En el constructor de **MiBoton**, se establece el color del botón con una llamada a **setBackground( )**. Los oyentes están todos instalados con simples llamadas a métodos.

La clase **RastrearEvento** contiene un **HashMap** para guardar las cadenas que representan el tipo de evento y **JTextFields** donde se guarda la información sobre el evento. Por supuesto, éstos podrían haberse creado de forma estática en vez de ponerlos en un **HashMap**, pero creo que estará de acuerdo en que es mucho más fácil de usar y cambiar. En particular, si se necesita añadir o retirar un nuevo tipo de evento en **RastrearEvento**, simplemente se añadirá o retirará un String en el array **evento** —todo lo demás sucede automáticamente.

Cuando se invoca a **informar( )** se le da el nombre del evento y el parámetro String del evento. Usa el **HashMap h** en la clase externa para buscar el **JTextField** asociado con ese nombre de evento y después coloca el parámetro String en ese campo.

Es divertido ejecutar este ejemplo puesto que realmente se puede ver lo que está ocurriendo con los eventos del programa.

Un catálogo de componentes Swing

Ahora que se entienden los gestores de disposición y el modelo de eventos, ya podemos ver cómo se pueden usar los componentes Swing. Esta sección es un viaje no exhaustivo por los componen-

tes de Swing y las facetas que probablemente se usarán la mayoría de veces. Cada ejemplo pretende ser razonablemente pequeño, de forma que se pueda tomar el código y usarlo en los programas que cada uno desarrolle.

Se puede ver fácilmente qué aspecto tiene cada uno de los ejemplos al ejecutarlo viendo las páginas HTML en el código fuente descargable para este capítulo.

Mantenga en mente:

1. La documentación HTML de *java.sun.com* contiene todas las clases y métodos Swing (aquí sólo se muestran unos pocos).

2. Debido a la convención de nombres usada en los eventos Swing, es bastante fácil adivinar cómo escribir e instalar un manipulador para un tipo particular de evento. Se puede usar el programa de búsqueda **MostrarAddListeners.java** visto antes en este capítulo para ayudar a investigar un componente particular.

3. Cuando las cosas se complican habría que pasar a un constructor de IGU.

Botones

Swing incluye varios tipos de botones. Todos los botones, casillas de verificación, botones de opción e incluso los elementos de menú se heredan de **AbstractButton** (que, dado que incluye elementos de menú, se habría llamado probablemente "AbstractChooser" o algo igualmente general). En breve veremos el uso de elementos de menú, pero el ejemplo siguiente muestra los distintos tipos de botones disponibles:

```
//: c13:Botones.java
// Varios botones Swing.
// <applet code=Botones
//  width=350 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.plaf.basic.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Botones extends JApplet {
  JButton jb = new JButton("JButton");
  BasicArrowButton
    arriba = new BasicArrowButton(
      BasicArrowButton.NORTH),
    abajo = new BasicArrowButton(
      BasicArrowButton.SOUTH),
    derecha = new BasicArrowButton(
```

```
        BasicArrowButton.EAST),
      izquierda = new BasicArrowButton(
        BasicArrowButton.WEST);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(jb);
    cp.add(new JToggleButton("JToggleButton"));
    cp.add(new JCheckBox("JCheckBox"));
    cp.add(new JRadioButton("JRadioButton"));
    JPanel jp = new JPanel();
    jp.setBorder(new TitledBorder("Direcciones"));
    jp.add(arriba);
    jp.add(abajo);
    jp.add(izquierda);
    jp.add(derecha);
    cp.add(jp);
  }
  public static void main(String[] args) {
    Console.run(new Buttons(), 350, 100);
  }
} ///:~
```

Éste comienza con el **BasicArrowButton** de **javax.swing.plaf.basic**, después continúa con los diversos tipos específicos de botones. Al ejecutar el ejemplo, se verá que el botón de conmutación guarda su última posición, dentro o fuera. Pero las casillas de verificación y los botones de opción se comportan exactamente igual simplemente pulsando para activarlos o desactivarlos (ambos se heredan de **JToggleButton**).

Grupos de botones

Si se desea que varios botones de opción se comporten en forma de "or exclusivo o XOR", hay que añadirlos a un "grupo de botones". Pero, como demuestra el ejemplo de debajo, se puede añadir cualquier **AbstractButton** a un **ButtonGroup**.

Para evitar repetir mucho código, este ejemplo usa la reflectividad para generar los grupos de distintos tipos de botones. Esto se ve en **hacerBPanel( )**, que crea un grupo de botones y un **JPanel**. El segundo parámetro a **hacerBPanel( )** es un array de **String**. Por cada **String**, se añade un botón de la clase indicada por el primer argumento al **Janel**:

```
//: c13:GrupoBotones.java
// Usa la reflectividad para crear grupos
// de diferentes tipos de GrupoBotones.
// <applet code=GrupoBotones
//  width=500 height=300></applet>
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.lang.reflect.*;
import com.bruceeckel.swing.*;

public class GrupoBotones extends JApplet {
  static String[] ids = {
    "June", "Ward", "Beaver",
    "Wally", "Eddie", "Lumpy",
  };
  static JPanel
  hacerBPanel(Class bClass, String[] ids) {
    ButtonGroup bg = new ButtonGroup();
    JPanel jp = new JPanel();
    String titulo = bClass.getName();
    titulo = titulo.substring(
      titulo.lastIndexOf('.') + 1);
    jp.setBorder(new TitledBorder(titulo));
    for(int i = 0; i < ids.length; i++) {
      AbstractButton ab = new JButton("Fallo");
      try {
        // Lograr el método constructor dinámico
        // que toma un argumento String:
        Constructor ctor = bClass.getConstructor(
          new Class[] { String.class });
        // Crear un objeto nuevo:
        ab = (AbstractButton)ctor.newInstance(
          new Object[]{ids[i]});
      } catch(Exception ex) {
        System.err.println("no se puede crear " +
          bClass);
      }
      bg.add(ab);
      jp.add(ab);
    }
    return jp;
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(hacerBPanel(JButton.class, ids));
    cp.add(hacerBPanel(JToggleButton.class, ids));
    cp.add(hacerBPanel(JCheckBox.class, ids));
    cp.add(hacerBPanel(JRadioButton.class, ids));
```

```
  }
  public static void main(String[] args) {
    Console.run(new GrupoBotones(), 500, 300);
  }
} ///:~
```

El título del borde se toma del nombre de la clase, eliminando la información de trayectoria. El **AbstractButton** se inicializa a **JButton**, que tiene la etiqueta "Fallo" por lo que si se ignora el mensaje de excepción, se seguirá viendo el problema en pantalla. El método **getConstructor( )** produce un objeto **Constructor** que toma el array de argumentos de los tipos del array **Class** pasado a **getConstructor( )**. Después todo lo que se hace es llamar a **newInstance( )**, pasándole un array de **Object** que contiene los parámetros actuales —en este caso, simplemente el **String** del array **ids**.

Esto añade un poco de complejidad a lo que es un proceso simple. Para lograr comportamiento XOR con botones, se crea un grupo de botones y se añade cada botón para el que se desea ese comportamiento en el grupo. Cuando se ejecuta el programa, se verá que todos los botones excepto **JButton** exhiben este comportamiento "or exclusivo".

Iconos

Se puede usar un **Icon** dentro de un **JLabel** o cualquier cosa heredada de **AbstractButton** (incluyendo **JButton**, **JCheckBox**, **JRadioButton**, y los distintos tipos de **JMenuItem**). Utilizar **Icons** con **JLabels** es bastante directo (se verá un ejemplo más adelante). El ejemplo siguiente explora todas las formas adicionales de usar **Icons** con botones y sus descendientes.

Se puede usar cualquier archivo **gif** que se desee, pero los que se usan en este ejemplo son parte de la distribución de código de este libro, disponible en *http://www.BruceEckel.com*. Para abrir un archivo e incorporar la imagen, simplemente se crea un **ImageIcon** y se le pasa el nombre del archivo. A partir de ese momento, se puede usar el **Icon** resultante en el programa.

Nótese que en este ejemplo, la información de trayectoria está codificada a mano; se necesitará cambiar la trayectoria para hacerla corresponder con la ubicación de los archivos de imágenes:

```
//: c13:Caras.java
// Comportamiento de Icon en JButtons.
// <applet code=Caras
//  width=250 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Caras extends JApplet {
  // La información de ruta siguiente es necesaria
  // para ejecutarse via un applet directamente desde el disco:
  static String ruta =
```

```
      "C:/aaa-TIJ2-distribution/code/c13/";
  static Icon[] caras = {
    new ImageIcon(rutas + "face0.gif"),
    new ImageIcon(rutas + "face1.gif"),
    new ImageIcon(rutas + "face2.gif"),
    new ImageIcon(rutas + "face3.gif"),
    new ImageIcon(rutas + "face4.gif"),
  };
  JButton
    jb = new JButton("JButton", caras[3]),
    jb2 = new JButton("Deshabilitar");
  boolean loco = false;
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    jb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(loco) {
          jb.setIcon(caras[3]);
          loco = false;
        } else {
          jb.setIcon(caras[0]);
          loco = true;
        }
        jb.setVerticalAlignment(JButton.TOP);
        jb.setHorizontalAlignment(JButton.LEFT);
      }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(caras[1]);
    jb.setPressedIcon(caras[2]);
    jb.setDisabledIcon(caras[4]);
    jb.setToolTipText("¡Yow!");
    cp.add(jb);
    jb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(jb.isEnabled()) {
          jb.setEnabled(false);
          jb2.setText("Habilitar");
        } else {
          jb.setEnabled(true);
          jb2.setText("Deshabilitar");
        }
      }
    });
```

```
    cp.add(jb2);
  }
  public static void main(String[] args) {
    Console.run(new Caras(), 400, 200);
  }
} ///:~
```

Se puede usar un **Icon** en muchos constructores, pero también se puede usar **setIcon( )** para añadir o cambiar un **Icon**. Este ejemplo también muestra cómo un **JButton** (o cualquier **AbstractButton**) puede establecer los distintos tipos de iconos que aparecen cuando le ocurren cosas a ese botón: cuando se presiona, se deshabilita o se "pasa sobre él" (el ratón se mueve sobre él sin hacer clic). Se verá que esto da a un botón una imagen animada genial.

Etiquetas de aviso

El ejemplo anterior añadía una "etiqueta de aviso" al botón. Casi todas las clases que se usen para crear interfaces de usuario se derivan de **JComponent**, que contiene un método denominado **setToolTipText(String)**. Por tanto, para casi todo lo que se coloque en un formulario, todo lo que se necesita hacer es decir (siendo **jc** el objeto de cualquier clase derivada **JComponent**):

```
jc.setToolTipText("Mi etiqueta");
```

y cuando el ratón permanece sobre ese **JComponent**, durante un periodo de tiempo predeterminado, aparecerá una pequeña caja con el texto junto al puntero del ratón.

Campos de texto

Este ejemplo muestra el comportamiento extra del que son capaces los **JTextFields**:

```
//: c13:CamposTexto.java
// Campos de texto y eventos Java.
// <applet code=CamposTexto width=375
// height=125></applet>
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class CamposTexto extends JApplet {
  JButton
    b1 = new JButton("Leer Texto"),
    b2 = new JButton("Poner Texto");
  JTextField
    t1 = new JTextField(30),
```

```
    t2 = new JTextField(30),
    t3 = new JTextField(30);
  String s = new String();
  DocumentoMayusculas
    ucd = new DocumentoMayusculas();
  public void init() {
    t1.setDocument(ucd);
    ucd.addDocumentListener(new T1());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    DocumentListener dl = new T1();
    t1.addActionListener(new T1A());
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(t1);
    cp.add(t2);
    cp.add(t3);
  }
  class T1 implements DocumentListener {
    public void changedUpdate(DocumentEvent e){}
    public void insertUpdate(DocumentEvent e){
      t2.setText(t1.getText());
      t3.setText("Texto: "+ t1.getText());
    }
    public void removeUpdate(DocumentEvent e){
      t2.setText(t1.getText());
    }
  }
  class T1A implements ActionListener {
    private int conteo = 0;
    public void actionPerformed(ActionEvent e) {
      t3.setText("t1 Action Event " + conteo++);
    }
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(t1.getSelectedText() == null)
        s = t1.getText();
      else
        s = t1.getSelectedText();
      t1.setEditable(true);
    }
  }
```

```
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      ucd.setUpperCase(false);
      t1.setText("Insertado por el Boton 2: " + s);
      ucd.setUpperCase(true);
      t1.setEditable(false);
    }
  }
  public static void main(String[] args) {
    Console.run(new CamposTexto(), 375, 125);
  }
}

class DocumentoMayusculas extends PlainDocument {
  boolean mayusculas = true;
  public void setUpperCase(boolean flag) {
    mayusculas = flag;
  }
  public void insertString(int offset,
    String string, AttributeSet attributeSet)
    throws BadLocationException {
      if(mayusculas)
        string = string.toUpperCase();
      super.insertString(offset,
        string, attributeSet);
  }
} ///:~
```

El **JTextField t3** se incluye como un lugar a reportar cuando se dispare el oyente del **JTextField t1**. Se verá que el oyente de la acción de un **JTextField** se dispara sólo al presionar la tecla "enter".

El **JTextField t1** tiene varios oyentes asignados. El **T1** es un **DocumentListener** que responde a cualquier cambio en el "documento" (en este caso los contenidos de **JTextField**). Copia automáticamente todo el texto de **t1** a **t2**. Además, el documento de **t1** se pone a una clase derivada de **PlainDocument**, llamada **DocumentoMayusculas**, que fuerza a que todos sus caracteres sean mayúsculas. Detecta automáticamente los espacios en blanco y lleva a cabo los borrados, ajustando los intercalados y gestionando todo como cabría esperar.

Bordes

JComponent contiene un método denominado **setBorder( )**, que permite ubicar varios bordes interesantes en cualquier componente visible. El ejemplo siguiente demuestra varios de los distintos bordes disponibles, utilizando un método denominado **showBorder( )** que crea un **JPanel** y pone el borde en cada caso. También usa RTTI para averiguar el nombre del borde que se está usando

(eliminando toda información de trayectoria), y pone después ese nombre en un **JLabel** en el medio del panel:

```
//: c13:Bordes.java
// Bordes Swing diferentes.
// <applet code=Bordes
//  width=500 height=300></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Bordes extends JApplet {
  static JPanel mostrarBorde(Border b) {
    JPanel jp = new JPanel();
    jp.setLayout(new BorderLayout());
    String nm = b.getClass().toString();
    nm = nm.substring(nm.lastIndexOf('.') + 1);
    jp.add(new JLabel(nm, JLabel.CENTER),
      BorderLayout.CENTER);
    jp.setBorder(b);
    return jp;
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.setLayout(new GridLayout(2,4));
    cp.add(mostrarBorde(new TitledBorder("Titulo")));
    cp.add(mostrarBorde(new EtchedBorder()));
    cp.add(mostrarBorde(new LineBorder(Color.blue)));
    cp.add(mostrarBorde(
      new MatteBorder(5,5,30,30,Color.green)));
    cp.add(mostrarBorde(
      new BevelBorder(BevelBorder.RAISED)));
    cp.add(mostrarBorde(
      new SoftBevelBorder(BevelBorder.LOWERED)));
    cp.add(mostrarBorde(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.red))));
  }
  public static void main(String[] args) {
    Console.run(new Bordes(), 500, 300);
  }
} ///:~
```

También se pueden crear bordes personalizados y ponerlos dentro de botones, etiquetas, etc. —cualquier cosa derivada de **JComponent**.

JScrollPanes

La mayoría de las veces simplemente se deseará dejar que un **JScrollPane** haga su trabajo, pero también se pueda controlar qué barras de desplazamiento están permitidas —la vertical, la horizontal, ambas o ninguna:

```
//: c13:JScrollPanes.java
// Controlando las barras de desplazamiento en un JScrollPane.
// <applet code=JScrollPanes width=300 height=725>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class JScrollPanes extends JApplet {
  JButton
    b1 = new JButton("Area Texto 1"),
    b2 = new JButton("Area Texto 2"),
    b3 = new JButton("Reemplazar Texto"),
    b4 = new JButton("Insertar Texto");
  JTextArea
    t1 = new JTextArea("t1", 1, 20),
    t2 = new JTextArea("t2", 4, 20),
    t3 = new JTextArea("t3", 1, 20),
    t4 = new JTextArea("t4", 10, 10),
    t5 = new JTextArea("t5", 4, 20),
    t6 = new JTextArea("t6", 10, 10);
  JScrollPane
    sp3 = new JScrollPane(t3,
      JScrollPane.VERTICAL_SCROLLBAR_NEVER,
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp4 = new JScrollPane(t4,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp5 = new JScrollPane(t5,
      JScrollPane.VERTICAL_SCROLLBAR_NEVER,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS),
    sp6 = new JScrollPane(t6,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
```

```
  class B1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t5.append(t1.getText() + "\n");
    }
  }
  class B2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t2.setText("Insertado por el Boton 2");
      t2.append(": " + t1.getText());
      t5.append(t2.getText() + "\n");
    }
  }
  class B3L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String s = " Reemplazo ";
      t2.replaceRange(s, 3, 3 + s.length());
    }
  }
  class B4L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t2.insert(" Insertado ", 10);
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Crear Bordes para los componentes:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 1, 1, Color.black);
    t1.setBorder(brd);
    t2.setBorder(brd);
    sp3.setBorder(brd);
    sp4.setBorder(brd);
    sp5.setBorder(brd);
    sp6.setBorder(brd);
    // Inicializar los oyentes y añadir componentes:
    b1.addActionListener(new B1L());
    cp.add(b1);
    cp.add(t1);
    b2.addActionListener(new B2L());
    cp.add(b2);
    cp.add(t2);
    b3.addActionListener(new B3L());
    cp.add(b3);
    b4.addActionListener(new B4L());
```

```
    cp.add(b4);
    cp.add(sp3);
    cp.add(sp4);
    cp.add(sp5);
    cp.add(sp6);
  }
  public static void main(String[] args) {
    Console.run(new JScrollPanes(), 300, 725);
  }
} ///:~
```

Utilizar argumentos distintos en el constructor **JScrollPane** permite controlar las barras de desplazamiento disponibles. Este ejemplo también adorna un poco los distintos elementos usando bordes.

Un minieditor

El control **JTextPane** proporciona un gran soporte a la edición sin mucho esfuerzo. El ejemplo siguiente hace un uso muy sencillo de esto, ignorando el grueso de la funcionalidad de la clase:

```
//: c13:PanelTexto.java
// El control JTextPane es un pequeño editor.
// <applet code=PanelTexto width=475 height=425>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class PanelTexto extends JApplet {
  JButton b = new JButton("Añadir Texto");
  JTextPane tp = new JTextPane();
  static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public void init() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        for(int i = 1; i < 10; i++)
          tp.setText(tp.getText() +
            sg.next() + "\n");
      }
    });
    Container cp = getContentPane();
    cp.add(new JScrollPane(tp));
```

```
      cp.add(BorderLayout.SOUTH, b);
   }
   public static void main(String[] args) {
      Console.run(new PanelTexto(), 475, 425);
   }
} ///:~
```

El botón simplemente añade texto generado al azar. La intención del **JTextPane** es permitir editar texto *in situ*, de forma que se verá que no hay método **append( )**. En este caso (hay que admitir que se trata de un uso pobre de las capacidades de **JTextPane**), debemos capturar el texto, modificarlo y volverlo a ubicar en su sitio utilizando **setText( )**.

Como se mencionó anteriormente, el comportamiento de la disposición por defecto de un *applet* es usar el **BorderLayout**. Si se añade algo al panel sin especificar más detalles, simplemente se rellena el centro del panel hasta los bordes. Sin embargo, si se especifica una de las regiones que le rodean (NORTH, SOUTH, EAST o WEST) como se hace aquí, el componente se encajará en esa región —en este caso, el botón se anidará abajo, en la parte inferior de la pantalla.

Fíjese en las facetas incluidas en **JTextPane**, como la envoltura automática de líneas. Usando la documentación JDK es posible buscar otras muchas facetas.

Casillas de verificación

Una casilla de verificación proporciona una forma sencilla de hacer una elección de tipo activado/desactivado; consiste en una pequeña caja y una etiqueta. La caja suele guardar una pequeña "x" (o alguna otra indicación que se establezca), o está vacía, en función de si el elemento está o no seleccionado.

Normalmente se creará un **JCheckBox** utilizando un constructor que tome la etiqueta como parámetro. Se puede conseguir y establecer su estado, y también la etiqueta si se desea leerla o cambiarla una vez creado el **JCheckBox**.

Siempre que se asigne valor o se limpie un **JCheckBox**, se da un evento que se puede capturar de manera análoga a como se hace con un botón, utilizando un **ActionListener**. El ejemplo siguiente usa un **JTextArea** para enumerar todas las casillas de verificación seleccionadas:

```
//: c13:CasillasVerificacion.java
// Uso de JCheckBox.
// <applet code=CasillasVerificacion width=200 height=200>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CasillasVerificacion extends JApplet {
```

```
  JTextArea t = new JTextArea(6, 15);
  JCheckBox
    cb1 = new JCheckBox("Check Box 1"),
    cb2 = new JCheckBox("Check Box 2"),
    cb3 = new JCheckBox("Check Box 3");
  public void init() {
    cb1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        rastrear("1", cb1);
      }
    });
    cb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        rastrear("2", cb2);
      }
    });
    cb3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        rastrear("3", cb3);
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(cb1);
    cp.add(cb2);
    cp.add(cb3);
  }
  void rastrear(String b, JCheckBox cb) {
    if(cb.isSelected())
      t.append("Box " + b + " Establecido\n");
    else
      t.append("Box " + b + " Limpiado\n");
  }
  public static void main(String[] args) {
    Console.run(new CasillasVerificacion(), 200, 200);
  }
} ///:~
```

El método **Rastrear( )** envía el nombre del **JCheckBox** seleccionado y su estado actual al **JTextArea** utilizando **append( )**, por lo que se verá una lista acumulativa de casillas de verificación seleccionadas y cuál es su estado.

Botones de opción

El concepto de botón de opción en programación de IGU proviene de las radios de coche preelectrónicas con botones mecánicos: cuando se oprimía un botón cualquier otro botón que estuviera pulsado, saltaba. Por consiguiente, permite forzar una elección única entre varias.

Todo lo que se necesita para establecer un grupo asociado de **JRadioButton**s es añadirlos a un **ButtonGroup** (se puede tener cualquier número de **ButtonGroup**s en un formulario). Uno de los botones puede tener su valor inicial por defecto a **true** (utilizando el segundo argumento del constructor). Si se intenta establecer más de un botón de opción a **true** sólo quedará con este valor el último al que se le asigne.

He aquí un ejemplo simple del uso de botones de opción. Nótese que se pueden capturar eventos de botones de opción, al igual que con los otros:

```
//: c13:BotonesOpcion.java
// Usando JRadioButtons.
// <applet code=BotonesOpcion
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BotonesOpcion extends JApplet {
  JTextField t = new JTextField(15);
  ButtonGroup g = new ButtonGroup();
  JRadioButton
    rb1 = new JRadioButton("uno", false),
    rb2 = new JRadioButton("dos", false),
    rb3 = new JRadioButton("tres", false);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      t.setText("Radio button " +
        ((JRadioButton)e.getSource()).getText());
    }
  };
  public void init() {
    rb1.addActionListener(al);
    rb2.addActionListener(al);
    rb3.addActionListener(al);
    g.add(rb1); g.add(rb2); g.add(rb3);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
```

```
      cp.add(rb1);
      cp.add(rb2);
      cp.add(rb3);
    }
    public static void main(String[] args) {
      Console.run(new BotonesOpcion(), 200, 100);
    }
} ///:~
```

Para mostrar el estado, se usa un campo de texto. Este campo se pone a no editable pues se usa para mostrar datos, no para recogerlos. Por consiguiente, es una alternativa al uso de una **JLabel**.

Combo boxes (listas desplegables)

Al igual que un grupo de botones de opción, una lista desplegable es una forma de obligar al usuario a seleccionar sólo un elemento a partir de un grupo de posibles elementos. Sin embargo, es una forma más compacta de lograrlo, y es más fácil cambiar los elementos de la lista sin soprender al usuario. (Se pueden cambiar botones de opción dinámicamente, pero esto tiende a ser demasiado visible.)

El **JComboBox** de Java no es como el cuadro combinado de Windows, que permite seleccionar a partir de una lista *o* tipo de la propia selección. Con un **JComboBox** se puede elegir uno y sólo un elemento de la lista. En el ejemplo siguiente, la **JComboBox** comienza con cierto número de entradas, añadiéndosele otras cuando se presiona un botón:

```
//: c13:CuadrosCombinados.java
// Usando listas desplegables.
// <applet code=CuadrosCombinados
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CuadrosCombinados extends JApplet {
  String[] descripcion = { "Bullicioso", "Obtuso",
    "Recalcitrante", "Brillante", "Somnoliento",
    "Temoroso", "Florido", "Putrefacto" };
  JTextField t = new JTextField(15);
  JComboBox c = new JComboBox();
  JButton b = new JButton("Añadir elementos");
  int conteo = 0;
  public void init() {
    for(int i = 0; i < 4; i++)
      c.addItem(descripcion[conteo++]);
    t.setEditable(false);
    b.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e){
          if(conteo < descripcion.length)
            c.addItem(descripcion[conteo++]);
        }
      });
      c.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          t.setText("indice: "+ c.getSelectedIndex()
            + "   " + ((JComboBox)e.getSource())
            .getSelectedItem());
        }
      });
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(t);
      cp.add(c);
      cp.add(b);
    }
    public static void main(String[] args) {
      Console.run(new CuadrosCombinados(), 200, 100);
    }
} ///:~
```

La **JTextField** muestra los "índices seleccionados", que es la secuencia del elemento actualmente seleccionado, así como la etiqueta del botón de opción.

Listas

Estas cajas son significativamente diferentes de las **JComboBox**, y no sólo en apariencia. Mientras que una **JComboBox** se despliega al activarla, una **JList** ocupa un número fijo de líneas en la pantalla todo el tiempo y no cambia. Si se desea ver los elementos de la lista, simplemente se invoca a **getSelectedValues( )**, que produce un array de **String** de los elementos seleccionados.

Una **JList** permite selección múltiple: si se hace control-clic en más de un elemento (manteniendo pulsada la tecla "control" mientras que se llevan a cabo varios clics de ratón) el elemento original sigue resaltado y se puede seleccionar tantos como se desee. Si se selecciona un elemento, y después se pulsa mayúsculas-clic en otro elemento, se seleccionan todos los elementos comprendidos entre ambos. Para eliminar un elemento de un grupo se puede hacer control-clic sobre él.

```
//: c13:Lista.java
// <applet code=Lista width=250
// height=375> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Lista extends JApplet {
  String[] sabores = { "Chocolate", "Fresa",
    "Fundido de Vanilla", "Galleta de Menta",
    "Moka con Almendras", "Amanecer de Ron",
    "Crema de Praline", "Pastel de Barro" };
  DefaultListModel elementos=new DefaultListModel();
  JList lst = new JList(elementos);
  JTextArea t = new JTextArea(sabores.length,20);
  JButton b = new JButton("Añadir elemento");
  ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(conteo < sabores.length) {
        lElementos.add(0, sabores[conteo++]);
      } else {
        // Deshabilitar, puesto que no hay
        // más sabores que añadir a la Lista
        b.setEnabled(false);
      }
    }
  };
  ListSelectionListener ll =
    new ListSelectionListener() {
      public void valueChanged(
        ListSelectionEvent e) {
          t.setText("");
          Object[] elementos=lst.getSelectedValues();
          for(int i = 0; i < elementos.length; i++)
            t.append(elementos[i] + "\n");
        }
    };
  int conteo = 0;
  public void init() {
    Container cp = getContentPane();
    t.setEditable(false);
    cp.setLayout(new FlowLayout());
    // Crear Bordes para los componentes:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 2, 2, Color.black);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Añadir los primeros cuatro elementos a la lista
    for(int i = 0; i < 4; i++)
```

```
        lElementos.addElement(sabores[conteo++]);
      // Añadir elementos al Panel Contenedor para ser mostrados
      cp.add(t);
      cp.add(lst);
      cp.add(b);
      // Registrar los oyentes de eventos
      lst.addListSelectionListener(ll);
      b.addActionListener(bl);
    }
    public static void main(String[] args) {
      Console.run(new Lista(), 250, 375);
    }
} ///:~
```

Cuando se presiona el botón, añade elementos a la parte *superior* de la lista (porque el segundo argumento de **addItem( )** es 0).

Podemos ver que también se han añadido bordes a las listas.

Si se desea poner un array de **Strings** en una **JList**, hay una solución mucho más simple: se pasa el array al constructor **JList**, y construye la lista automáticamente. La única razón para usar el "modelo lista" en el ejemplo de arriba es que la lista puede ser manipulada durante la ejecución del programa.

Las **JLists** no proporcionan soporte directo para el desplazamiento. Por supuesto, todo lo que hay que hacer es envolver la **JList** en un **JScrollPane** y se logrará la gestión automática de todos los detalles.

Paneles Tabulados

El **JTabbedPane** permite crear un "diálogo con lengüetas", que tiene lengüetas de carpetas de archivos ejecutándose a través de un borde, y todo lo que hay que hacer es presionar una lengüeta para presentar un diálogo diferente:

```
//: c13:PanelTabulado1.java
// Demuestra el Panel Tabulado.
// <applet code=PanelTabulado1
// width=350 height=200> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class PanelTabulado1 extends JApplet {
  String[] sabores = { "Chocolate", "Fresa",
    "Fundido de Vainilla", "Galleta de Menta",
```

```
      "Moka con Almendras", "Amanecer de Ron",
      "Crema de Praline", "Pastel de Barro" };
  JTabbedPane tabs = new JTabbedPane();
  JTextField txt = new JTextField(20);
  public void init() {
    for(int i = 0; i < sabores.length; i++)
      tabs.addTab(sabores[i],
        new JButton("Panel Tabulado " + i));
    tabs.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        txt.setText("Tab seleccionado: " +
          tabs.getSelectedIndex());
      }
    });
    Container cp = getContentPane();
    cp.add(BorderLayout.SOUTH, txt);
    cp.add(tabs);
  }
  public static void main(String[] args) {
    Console.run(new PanelTabulado1(), 350, 200);
  }
} ///:~
```

En Java, el uso de algún tipo de mecanismo de "paneles tabulados" es bastante importante pues en programación de *applets* se suele intentar desmotivar el uso de diálogos emergentes añadiendo automáticamente un pequeño aviso a cualquier diálogo emergente de un *applet.*

Cuando se ejecute el programa se verá que el **JTabbedPane** comprime las lengüetas si hay demasiadas, de forma que quepan en una fila. Podemos ver esto redimensionando la ventana al ejecutar el programa desde la línea de comandos de la consola.

Cajas de mensajes

Los entornos de ventanas suelen contener un conjunto estándar de cajas de mensajes que permiten enviar información al usuario rápidamente, o capturar información proporcionada por éste. En Swing, estas cajas de mensajes se encuentran en **JOptionPane**. Hay muchas posibilidades diferentes (algunas bastante sofisticadas), pero las que más habitualmente se usan son probablemente el diálogo mensaje y el diálogo confirmación, invocados usando el **static JOptionPanel.showMessageDialog( )** y **JOptionPane.showConfirmDialog( )**. El ejemplo siguiente muestra un subconjunto de las cajas de mensajes disponibles con **JOptionPane**:

```
//: c13:CajasMensajes.java
// Demuestra JOptionPane.
// <applet code=CajasMensajes
// width=200 height=150> </applet>
import javax.swing.*;
```

```
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CajasMensajes extends JApplet {
  JButton[] b = { new JButton("Alerta"),
    new JButton("Si/No"), new JButton("Color"),
    new JButton("Entrada"), new JButton("3 Vals")
  };
  JTextField txt = new JTextField(15);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String id =
        ((JButton)e.getSource()).getText();
      if(id.equals("Alerta"))
        JOptionPane.showMessageDialog(null,
          "¡Ahí hay un fallo!", "¡Hey!",
          JOptionPane.ERROR_MESSAGE);
      else if(id.equals("Si/No"))
        JOptionPane.showConfirmDialog(null,
          "o no", "elegir si",
          JOptionPane.YES_NO_OPTION);
      else if(id.equals("Color")) {
        Object[] opciones = { "Rojo", "Verde" };
        int sel = JOptionPane.showOptionDialog(
          null, "¡Elija un Color!", "Precaucion",
          JOptionPane.DEFAULT_OPTION,
          JOptionPane.WARNING_MESSAGE, null,
          opciones, opciones[0]);
          if(sel != JOptionPane.CLOSED_OPTION)
            txt.setText(
              "Color Seleccionado: " + opciones[sel]);
      } else if(id.equals("Entrada")) {
        String val = JOptionPane.showInputDialog(
            "¿Cuantos dedos ves?");
        txt.setText(val);
      } else if(id.equals("3 Vals")) {
        Object[] selecciones = {
          "Primero", "Segundo", "Tercero" };
        Object val = JOptionPane.showInputDialog(
          null, "Elegir una", "Entrada",
          JOptionPane.INFORMATION_MESSAGE,
          null, selecciones, selecciones[0]);
        if(val != null)
          txt.setText(
```

```
              val.toString());
        }
      }
    };
    public void init() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      for(int i = 0; i < b.length; i++) {
        b[i].addActionListener(al);
        cp.add(b[i]);
      }
      cp.add(txt);
    }
    public static void main(String[] args) {
      Console.run(new CajasMensajes(), 200, 200);
    }
} ///:~
```

Para poder escribir un único **ActionListener**, he usado un enfoque con riesgo, comprobando las etiquetas **String** de los botones. El problema de todo esto es que es fácil provocar algún fallo con las etiquetas en el uso de mayúsculas, y este fallo podría ser difícil de localizar.

Nótese que **showOptionDialog( )** y **showInputDialog( )** proporcionan objetos de retorno que contienen el valor introducido por el usuario.

Menús

Cada componente capaz de guardar un menú, incluyendo **JApplet**, **JFrame**, **JDialog** y sus descendientes, tiene un método **setMenuBar( )** que acepta un **JMenuBar** (sólo se puede tener un **JMenuBar** en un componente particular). Se añaden **JMenus** al **JMenuBar**, y **JMenuItems** a los **JMenus**. Cada **JMenuItem** puede tener un **ActionListener** asociado, que se dispara al seleccionar ese elemento del menú.

A diferencia de un sistema basado en el uso de recursos, con Java y Swing hay que ensamblar a mano todos los menús en código fuente. He aquí un ejemplo de menú sencillo:

```
//: c13:MenusSimples.java
// <applet code=MenusSimples
// width=200 height=75> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
public class MenusSimples extends JApplet {
  JTextField t = new JTextField(15);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      t.setText(
        ((JMenuItem)e.getSource()).getText());
    }
  };
  JMenu[] menus = { new JMenu("Winken"),
    new JMenu("Blinken"), new JMenu("Nod") };
  JMenuItem[] elementos = {
    new JMenuItem("Fee"), new JMenuItem("Fi"),
    new JMenuItem("Fo"),  new JMenuItem("Zip"),
    new JMenuItem("Zap"), new JMenuItem("Zot"),
    new JMenuItem("Olly"), new JMenuItem("Oxen"),
    new JMenuItem("Free") };
  public void init() {
    for(int i = 0; i < elementos.length; i++) {
      elementos[i].addActionListener(al);
      menus[i%3].add(elementos[i]);
    }
    JMenuBar mb = new JMenuBar();
    for(int i = 0; i < menus.length; i++)
      mb.add(menus[i]);
    setJMenuBar(mb);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
  }
  public static void main(String[] args) {
    Console.run(new MenusSimples(), 200, 75);
  }
} ///:~
```

El uso del operador módulo en "**i%3**" distribuye los elementos de menú entre los tres **JMenus**. Cada **JMenuItem** debe tener un **ActionListener** adjunto; aquí se usa el mismo **ActionListener** en todas partes pero generalmente será necesario uno individual para cada **JMenuItem**.

JMeuItem hereda de **AbstractButton**, por lo que tiene algunos comportamientos propios de los botones. Por sí mismo, proporciona un elemento que se puede ubicar en un menú desplegable. También hay tres tipos heredados de **JMenuItem**: **JMenu** para albergar otros **JMenuItems** (de forma que se pueden tener menús en cascada), **JCheckBoxMenuItem**, que produce una marca que indica si se ha seleccionado o no ese elemento de menú, y **JRadioButtonMenuItem**, que contiene un botón de opción.

Como ejemplo más sofisticado, he aquí el de los sabores de helado de nuevo, utilizado para crear menús. Este ejemplo también muestra menús en cascada, mnemónicos de teclado, **JCheckBox MenuItems**, y la forma de cambiar menús dinámicamente:

```
//: c13:Menus.java
// Submenús, elementos de menú casillas de verificación, menús intercambiables,
// mnemónicos (atajos) y comandos de acción.
// <applet code=Menus width=300
// height=100> </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Menus extends JApplet {
  String[] sabores = { "Chocolate", "Fresa",
    "Fundido de Vainilla", "Galleta de Menta",
    "Moka con Almendras", "Amanecer de Ron",
    "Crema de Praline", "Pastel de Barro" };
  JTextField t = new JTextField("No sabor", 30);
  JMenuBar mb1 = new JMenuBar();
  JMenu
    f = new JMenu("Fichero"),
    m = new JMenu("Sabores"),
    s = new JMenu("Seguridad");
  // Enfoque alternativo:
  JCheckBoxMenuItem[] seguridad = {
    new JCheckBoxMenuItem("Guardar"),
    new JCheckBoxMenuItem("Ocultar")
  };
  JMenuItem[] archivo = {
    new JMenuItem("Abrir"),
  };
  // Una segunda barra de menú a la que cambiar a:
  JMenuBar mb2 = new JMenuBar();
  JMenu fooBar = new JMenu("fooBar");
  JMenuItem[] otro = {
    // Añadir un atajo de menú (mnemónico) es muy
    // simple, pero sólo los JMenuItems pueden tenerlos en
    // sus constructores:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
    // Sin atajo:
    new JMenuItem("Baz"),
  };
```

```
JButton b = new JButton("Intercambiar Menus");
class BL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    JMenuBar m = getJMenuBar();
    setJMenuBar(m == mb1 ? mb2 : mb1);
    validate(); // Refrescar el frame
  }
}
class ML implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    JMenuItem objetivo = (JMenuItem)e.getSource();
    String comandoAccion =
      objetivo.getActionCommand();
    if(comandoAccion.equals("Abrir")) {
      String s = t.getText();
      boolean elegido = false;
      for(int i = 0; i < sabores.length; i++)
        if(s.equals(sabores[i])) elegido = true;
      if(!elegido)
        t.setText("¡Elegir un sabor primero!");
      else
        t.setText("Abriendo "+ s +". Mmm, mm!");
    }
  }
}
class FL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    JMenuItem objetivo = (JMenuItem)e.getSource();
    t.setText(objetivo.getText());
  }
}
// Alternativamente, se puede crear una clase
// diferente por cada MenuItem diferente. Después no hay
// que averiguar cuál es:
class FooL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    t.setText("Foo seleccionado");
  }
}
class BarL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    t.setText("Bar seleccionado");
  }
}
class BazL implements ActionListener {
```

```
    public void actionPerformed(ActionEvent e) {
      t.setText("Baz seleccionado");
    }
  }
  class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      JCheckBoxMenuItem objetivo =
        (JCheckBoxMenuItem)e.getSource();
      String comandoAccion =
        target.getActionCommand();
      if(comandoAccion.equals("Guardar"))
        t.setText("¡Guardar el helado! " +
          "Guardando esta a " + objetivo.getState());
      else if(comandoAccion.equals("Ocultar"))
        t.setText("¡Esconder el Helado! " +
          "¿Esta frio? " + objetivo.getState());
    }
  }
  public void init() {
    ML ml = new ML();
    CMIL cmil = new CMIL();
    seguridad[0].setActionCommand("Guardar");
    seguridad[0].setMnemonic(KeyEvent.VK_G);
    seguridad[0].addItemListener(cmil);
    seguridad[1].setActionCommand("Ocultar");
    seguridad[0].setMnemonic(KeyEvent.VK_H);
    seguridad[1].addItemListener(cmil);
    otro[0].addActionListener(new FooL());
    otro[1].addActionListener(new BarL());
    otro[2].addActionListener(new BazL());
    FL fl = new FL();
    for(int i = 0; i < sabores.length; i++) {
      JMenuItem mi = new JMenuItem(sabores[i]);
      mi.addActionListener(fl);
      m.add(mi);
      // Añadir separadores a intervalos:
      if((i+1) % 3 == 0)
        m.addSeparator();
    }
    for(int i = 0; i < seguridad.length; i++)
      s.add(seguridad[i]);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < archivo.length; i++) {
```

```
        archivo[i].addActionListener(fl);
        f.add(archivo[i]);
      }
      mb1.add(f);
      mb1.add(m);
      setJMenuBar(mb1);
      t.setEditable(false);
      Container cp = getContentPane();
      cp.add(t, BorderLayout.CENTER);
      // Establecer el sistema para menús intercambiables:
      b.addActionListener(new BL());
      b.setMnemonic(KeyEvent.VK_S);
      cp.add(b, BorderLayout.NORTH);
      for(int i = 0; i < otro.length; i++)
        fooBar.add(otro[i]);
      fooBar.setMnemonic(KeyEvent.VK_B);
      mb2.add(fooBar);
    }
    public static void main(String[] args) {
      Console.run(new Menus(), 300, 100);
    }
} ///:~
```

En este programa hemos ubicado los elementos del menú en arrays y después se recorre cada array invocando a **add( )** por cada **JMenuItem**. De esta forma se logra que la adición o substracción de un elemento de menú sea bastante menos tediosa.

Este programa crea no uno, sino dos **JMenuBar** para demostrar que se pueden intercambiar barras de menú activamente mientras se ejecuta el programa. Se puede ver cómo un **JMenuBar** está hecho de **JMenus**, y cada **JMenu** está hecho de **JMenuItems**, **JCheckBoxMenuItems**, o incluso otros **JMenus** (logrando así submenús). Cuando se ensambla un **JMenuBar**, éste puede instalarse en el programa actual con el método **setJMenuBar( )**. Nótese que al presionar el botón, se comprueba qué menú está actualmente instalado invocando a **getJMenuBar( )**, y pone la otra barra de menú en su sitio.

Al probar "Abrir" nótese que el deletreo y el uso de mayúsculas es crítico, pero Java no señala ningún error si no hay coincidencia exacta con "Abrir". Este tipo de comparación de cadenas de caracteres es una fuente de errores de programación.

La comprobación y la no comprobación de los elementos de menú se lleva a cabo automáticamente. El código que gestiona los **JCheckBoxMenuItems** muestran dos formas de determinar lo que se comprobó: la comprobación de cadenas de caracteres (que, como se mencionó arriba, no es un enfoque muy seguro, aunque se verá a menudo) y la coincidencia de todos los objetos destino de eventos. Como se ha mostrado, el método **getState( )** puede usarse para revelar el estado. También se puede cambiar el estado de un **JCheckBoxMenuItem** con **setState( )**.

Los eventos de los menús son un poco inconsistentes y pueden conducir a confusión: los **JMenuItems** usan **ActionListeners**, pero los **JCheckboxMenuItems** usan **ItemListeners**. Los ob-

jetos **JMenu** también pueden soportar **ActionListeners**, pero eso no suele ser de ayuda. En general, se adjuntarán oyentes a cada **JMenuItem**, **JCheckBoxMenuItem** o **JRadioButtonMenuItem**, pero el ejemplo muestra **ItemListeners** y **ActionListeners** adjuntados a los distintos componentes menú.

Swing soporta mnemónicos o "atajos de teclado", de forma que se puede seleccionar cualquier cosa derivada de **AbstractButton** (botón, elemento de menú, etc.) utilizando el teclado en vez del ratón. Éstos son bastante simples: para **JMenuItem** se puede usar el constructor sobrecargado que toma como segundo argumento el identificador de la clave. Sin embargo, la mayoría de **AbstractButtons** no tiene constructores como éste, por lo que la forma más general de solucionar el problema es usar el método **setMnemonic( )**. El ejemplo de arriba añade mnemónicos al botón y a algunos de los elementos de menús; los indicadores de atajos aparecen automáticamente en los componentes.

También se puede ver el uso de **setActionCommand( )**. Éste parece un poco extraño porque en cada caso el "comando de acción" es exactamente el mismo que la etiqueta en el componente del menú. ¿Por qué no usar simplemente la etiqueta en vez de esta cadena de caracteres alternativa? El problema es la internacionalización. Si se vuelve a direccionar el programa a otro idioma, sólo se deseará cambiar la etiqueta del menú, y no cambiar el código (lo cual podría sin duda introducir nuevos errores). Por ello, para que esto sea sencillo para el código que comprueba la cadena de texto asociada a un componente de menú, se puede mantener inmutable el "comando de acción" a la vez que se puede cambiar la etiqueta de menú. Todo el código funciona con el "comando de acción", pero no se ve afectada por los cambios en las etiquetas de menú. Nótese que en este programa, no se examinan los comandos de acción de todos los componentes del menú, por lo que los no examinados no tienen su comando de acción.

La mayoría del trabajo se da en los oyentes. **BL** lleva a cabo el intercambio de **JMenuBar**. En **ML**, se toma el enfoque de "averigua quién llama" logrando la fuente del **ActionEvent** y convirtiéndola en un **JMenuItem**, consiguiendo después la cadena de caracteres del comando de acción para pasarlo a través de una sentencia **if** en cascada.

El oyente **FL** es simple, incluso aunque esté gestionando todos los sabores distintos del menú de sabores. Este enfoque es útil para tomar el enfoque usado con **FooL**, **BarL** y **BazL**, en los que sólo se les ajunta un componente menú de forma que no es necesaria ninguna lógica extra de detección y se sabe exactamente quién invocó al oyente. Incluso con la profusión de clases generadas de esta manera, el código interno tiende a ser menor y el proceso es más a prueba de torpes.

Se puede ver que el código de menú se vuelve largo y complicado rápidamente. Éste es otro caso en el que la solución apropiada es usar un constructor de IGU. Una buena herramienta también gestionará el mantenimiento de menús.

Menús emergentes

La forma más directa de implementar un **JPopupMenu** es crear una clase interna que extienda **MouseAdapter**, después añadir un objeto de esa clase interna a cada componente que se desea para producir un comportamiento emergente:

```
//: c13:Emergente.java
// Creando menús emergentes con Swing.
// <applet code=Emergente
//  width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Emergente extends JApplet {
  JPopupMenu emergente = new JPopupMenu();
  JTextField t = new JTextField(10);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText(
          ((JMenuItem)e.getSource()).getText());
      }
    };
    JMenuItem m = new JMenuItem("Aquí");
    m.addActionListener(al);
    emergente.add(m);
    m = new JMenuItem("Yon");
    m.addActionListener(al);
    emergente.add(m);
    m = new JMenuItem("Afar");
    m.addActionListener(al);
    emergente.add(m);
    emergente.addSeparator();
    m = new JMenuItem("Permanecer aqui");
    m.addActionListener(al);
    emergente.add(m);
    OyenteEmergente pl = new OyenteEmergente();
    addMouseListener(pl);
    t.addMouseListener(pl);
  }
  class OyenteEmergente extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      quizasMostrarEmergente(e);
    }
    public void mouseReleased(MouseEvent e) {
      quizasMostrarEmergente(e);
```

```
      }
      private void quizasMostrarEmergente(MouseEvent e) {
         if(e.isPopupTrigger()) {
            emergente.show(
               e.getComponent(), e.getX(), e.getY());
         }
      }
   }
   public static void main(String[] args) {
      Console.run(new Emergente(), 300, 200);
   }
} ///:~
```

Se añade el mismo **ActionListener** a cada **JMenuItem**, de forma que tome el texto de la etiqueta de menú y lo inserte en el **JTextField**.

Generación de dibujos

En un buen marco de trabajo de IGU, la generación de dibujos debería ser razonablemente sencilla —y lo es, en la biblioteca Swing. El problema del ejemplo de dibujo es que los cálculos que determinan dónde van las cosas son bastante más complicados que las rutinas a las llamadas de generación de dibujos, y estos cálculos suelen mezclarse junto con las llamadas a dibujos de forma que puede parecer que la interfaz es más complicada que lo que es en realidad.

Por simplicidad, considérese el problema de representar datos en la pantalla —aquí, los datos los proporcionará el método **Math.sin( )** incluido que es la función matemática seno. Para hacer las cosas un poco más interesantes, y para demostrar más allá lo fácil que es utilizar componentes Swing, se puede colocar un deslizador en la parte de abajo del formulario para controlar dinámicamente el número de ondas cíclicas sinoidales que se muestran. Además, si se redimensiona la ventana, se verá que la onda seno se reajusta al nuevo tamaño de la ventana.

Aunque se puede pintar cualquier **JComponent**, y usarlo, por consiguiente, como lienzo, si simplemente se desea una superficie de dibujo, generalmente se heredará de un **JPanel** (es el enfoque más directo). Sólo hay que superponer el método **paintComponent( )**, que se invoca siempre que hay que repintar ese componente (generalmente no hay que preocuparse por esto pues se encarga Swing). Cuando es invocado, Swing le pasa un objeto **Graphics**, que puede usarse para dibujar o pintar en la superficie.

En el ejemplo siguiente, toda la inteligencia de pintado está en la clase **DibujarSeno**; la clase **OndaSeno** simplemente configura el programa y el control de deslizamiento. Dentro de **DibujarSeno**, el método **establecerCiclos( )** proporciona la posiblidad de permitir a otro objeto —en este caso el control de deslizamientos— controlar el número de ciclos.

```
//: c13:OndaSeno.java
// Dibujando con Swing, usando un JSlider.
// <applet code=OndaSeno
```

```
//  width=700 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class DibujarSeno extends JPanel {
  static final int FACTORESCALADO = 200;
  int ciclos;
  int puntos;
  double[] senos;
  int[] pts;
  DibujarSeno() { establecerCiclos(5); }
  public void establecerCiclos(int numCiclos) {
    ciclos = numCiclos;
    puntos = FACTORESCALADO * ciclos * 2;
    senos = new double[puntos];
    pts = new int[puntos];
    for(int i = 0; i < puntos; i++) {
      double radianes = (Math.PI/FACTORESCALADO) * i;
      senos[i] = Math.sin(radianes);
    }
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int anchuraMax = getWidth();
    double pasoH = (double)anchuraMax/(double)puntos;
    int alturaMax = getHeight();
    for(int i = 0; i < puntos; i++)
      pts[i] = (int)(senos[i] * alturaMax/2 * .95
                     + alturaMax/2);
    g.setColor(Color.red);
    for(int i = 1; i < puntos; i++) {
      int x1 = (int)((i - 1) * pasoH);
      int x2 = (int)(i * pasoH);
      int y1 = pts[i-1];
      int y2 = pts[i];
      g.drawLine(x1, y1, x2, y2);
    }
  }
}

public class OndaSeno extends JApplet {
  DibujarSeno senos = new DibujarSeno();
```

```
  JSlider ciclos = new JSlider(1, 30, 5);
  public void init() {
    Container cp = getContentPane();
    cp.add(senos);
    ciclos.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        senos.establecerCiclos(
          ((JSlider)e.getSource()).getValue());
      }
    });
    cp.add(BorderLayout.SOUTH, ciclos);
  }
  public static void main(String[] args) {
    Console.run(new OndaSeno(), 700, 400);
  }
} ///:~
```

Todos los miembros de datos y arrays se usan en el cálculo de los puntos de la onda senoidal: **ciclos** indica el número de ondas senoidales completas deseadas; **puntos** contiene el número total de puntos a dibujar, **senos** contiene los valores de la función seno, y **pts** contiene las coordenadas *y* de los puntos a dibujar en el **JPanel**. El método **establecerCiclos( )** crea los arrays de acuerdo con el número de puntos deseados y rellena el array **senos** con números. Llamar a **repaint( )**, fuerza a **establecerCiclos( )** a que se invoque a **paintComponent( )**, de forma que se lleven a cabo el resto de cálculos y redibujado.

Lo primero que hay que hacer al superponer **paintComponent( )** es invocar a la versión de la clase base del método. Después se puede hacer lo que se desee; normalmente, esto implica usar los métodos **Graphics** que se pueden encontrar en la documentación de **java.awt.Graphics** (en la documentación HTML de *http://java.sun.com*) para dibujar y pintar píxeles en el **JPanel**. Aquí, se puede ver que casi todo el código está involucrado en llevar a cabo los cálculos; de hecho, las dos únicas llamadas a métodos que manipulan la pantalla son **setColor( )** y **drawLine( )**. Probablemente se tendrá una experiencia similar al crear programas que muestren datos gráficos —se invertirá la mayor parte del tiempo en averiguar qué es lo que se desea dibujar, mientras que el proceso de dibujado en sí será bastante simple.

Cuando creamos este programa, la mayoría del tiempo se invirtió en mostrar la onda seno en la pantalla. Una vez que lo logramos, pensamos que sería bonito poder cambiar dinámicamente el número de ciclos. Mis experiencias programando al intentar hacer esas cosas en otros lenguajes, me hicieron dudar un poco antes de acometerlo, pero resultó ser la parte más fácil del proyecto. Creamos un **JSlider** (los argumentos son valores más a la izquierda del **JSlider**, los de más a la derecha y el valor de comienzo, respectivamente, pero también hay otros constructores) y lo volcamos en el **JApplet**. Después miramos en la documentación HTML y descubrimos que el único oyente era **addChangeListener**, que fue disparado siempre que se cambiase el *deslizador* lo suficiente como para producir un valor diferente. El único método para esto era el **stateChanged( )**, de nombre obvio, que proporcionó un objeto **ChangeEvent**, de forma que podía mirar hacia atrás a la fuente del cambio y encontrar el nuevo valor. Llamando a e**stablecerCiclos( )** del objeto **senos**, se incorporó el nuevo valor y se redibujó el **JPanel**.

En general, se verá que la mayoría de los problemas Swing se pueden solucionar siguiendo un proceso similar, y se averiguará que suele ser bastante simple, incluso si nunca antes se ha usado un componente particular.

Si el problema es más complejo, hay otras alternativas de dibujo más sofisticadas, incluyendo componentes JavaBeans de terceras partes y el API Java 2D. Estas soluciones se escapan del alcance de este libro, pero habría que investigarlo cuando el código de dibujo se vuelve demasiado oneroso.

Cajas de diálogo

Una caja de diálogo es una ventana que saca otra ventana. Su propósito es tratar con algún aspecto específico sin desordenar la ventana original con esos detalles. Las cajas de diálogo se usan muy intensivamente en entornos de programación con ventanas, pero se usan menos frecuentemente en los *applets*.

Para crear una caja de diálogo, se hereda de **JDialog**, que es simplemente otra clase de **Window**, como **JFrame**. Un **JDialog** tiene un gestor de disposiciones (por defecto **BorderLayout**) y se le añaden oyentes para que manipulen eventos. Una diferencia significativa al llamar a **windowClosing( )** es que no se desea apagar la aplicación. En vez de esto, se liberan los recursos usados por la ventana de diálogos llamando a **dispose( )**. He aquí un ejemplo muy sencillo:

```
//: c13:Dialogos.java
// Creando y usando Cajas de Diálogo.
// <applet code=Dialogos width=125 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class MiDialogo extends JDialog {
  public MiDialogo(JFrame parent) {
    super(parent, "Mi dialogo", true);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JLabel("He aqui mi dialogo"));
    JButton ok = new JButton("OK");
    ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dispose(); // Cierra el diálogo
      }
    });
    cp.add(ok);
    setSize(150,125);
```

```
  }
}

public class Dialogos extends JApplet {
  JButton b1 = new JButton("Caja de dialogo");
  MiDialogo dlg = new MiDialogo(null);
  public void init() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dlg.show();
      }
    });
    getContentPane().add(b1);
  }
  public static void main(String[] args) {
    Console.run(new Dialogos(), 125, 75);
  }
} ///:~
```

Una vez que se crea el **JDialog**, se debe llamar al método **show( )** para mostrarlo y activarlo. Para que se cierre el diálogo hay que llamar a **dispose( )**.

Veremos que cualquier cosa que surja de un objeto, incluyendo las cajas de diálogo, "no es de confianza". Es decir, se obtienen advertencias al usarlas. Esto es porque, en teoría, sería posible confundir al usuario y hacerle pensar que está tratando con una aplicación nativa regular y hacer que tecleen el número de su tarjeta de crédito que viajará después por la Web. Un *applet* siempre viaja adjunto a una página web, y será visible desde un navegador, mientas que las cajas de diálogo se desasocian —por lo que sería posible, en teoría. Como resultado no suele ser frecuente ver *applets* que hagan uso de cajas de diálogo.

El ejemplo siguiente es más complejo; la caja de diálogo consta de una rejilla (usando **GridLayout**) de un tipo especial de botón definido como clase **BotonToe**. Este botón dibuja un marco en torno a sí mismo y, dependiendo de su estado, un espacio en blanco, una "x" o una "o" en el medio. Comienza en blanco y después, dependiendo de a quién le toque, cambia a "x" o a "o". Sin embargo, también cambiará entre "x" y "o" al hacer clic en el botón. (Esto convierte el concepto tic-tac-toe en sólo un poco más asombroso de lo que ya es.) Además, se puede establecer que la caja de diálogo tenga un número cualquiera de filas y columnas, cambiando los números de la ventana de aplicación principal.

```
//: c13:TicTacToe.java
// Demostración de las cajas de diálogo
// y creación de componentes propios.
// <applet code=TicTacToe
//  width=200 height=100></applet>
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class TicTacToe extends JApplet {
  JTextField
    filas = new JTextField("3"),
    cols = new JTextField("3");
  static final int BLANCO = 0, XX = 1, OO = 2;
  class DialogoToe extends JDialog {
    int turno = XX; // Comenzar poniendo a "x"
    // w = número de celdas de ancho
    // h = número de celdas de alto
    public DialogoToe(int w, int h) {
      setTitle("El juego en si mismo");
      Container cp = getContentPane();
      cp.setLayout(new GridLayout(w, h));
      for(int i = 0; i < w * h; i++)
        cp.add(new BotonToe());
      setSize(w * 50, h * 50);
      // Diálogo de cierre de JDK 1.3:
      //#setDefaultCloseOperation(
      //#  DISPOSE_ON_CLOSE);
      // Diálogo de cierre de JDK 1.2:
      addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e){
          dispose();
        }
      });
    }
    class BotonToe extends JPanel {
      int estado = BLANCO;
      public BotonToe() {
        addMouseListener(new ML());
      }
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int x1 = 0;
        int y1 = 0;
        int x2 = getSize().width - 1;
        int y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int ancho = x2/2;
        int alto = y2/2;
```

```
        if(estado == XX) {
          g.drawLine(x1, y1,
            x1 + ancho, y1 + alto);
          g.drawLine(x1, y1 + alto,
            x1 + ancho, y1);
        }
        if(estado == OO) {
          g.drawOval(x1, y1,
            x1 + ancho/2, y1 + alto/2);
        }
      }
      class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
          if(estado == BLANK) {
            estado = turno;
            turno = (turno == XX ? OO : XX);
          }
          else
            estado = (estado == XX ? OO : XX);
          repaint();
        }
      }
    }
  }
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JDialog d = new DialogoToe(
        Integer.parseInt(pilas.getText()),
        Integer.parseInt(cols.getText()));
      d.setVisible(true);
    }
  }
  public void init() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(2,2));
    p.add(new JLabel("Filas", JLabel.CENTER));
    p.add(Pilas);
    p.add(new JLabel("Columnas", JLabel.CENTER));
    p.add(cols);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    JButton b = new JButton("Comenzar");
    b.addActionListener(new BL());
    cp.add(b, BorderLayout.SOUTH);
  }
```

```
  public static void main(String[] args) {
    Console.run(new TicTacToe(), 200, 100);
  }
} ///:~
```

Dado que los **statics** sólo pueden estar en el nivel externo de la clase, las clases internas no pueden tener datos **static** o clases internas **static**.

El método **paintComponent( )** dibuja el cuadrado alrededor del panel y la "x" o la "o". Esto implica muchos cálculos tediosos pero es directo.

El **MouseListener** captura los eventos de ratón, y comprueba en primer lugar si el panel ha escrito algo. Si no, se invoca a la ventana padre para averiguar a quién le toca, y qué se usa para establecer el estado del **BotonToe**. Vía el mecanismo de clases internas, posteriormente el **BotonToe** vuelve al padre y cambia el turno. Si el botón ya está mostrando una "x" o una "o", se cambia. En estos cálculos se puede ver el uso del "if-else" ternario descrito en el Capítulo 3. Después de un cambio de estado, se repinta el **BotonToe**.

El constructor de **DialogoToe** es bastante simple: añade en un **GridLayout** tantos botones como se solicite, y después lo redimensiona a 50 píxeles de lado para cada botón.

TicTacToe da entrada a toda la aplicación creando los **JTextFields** (para la introducción de las filas y columnas de la rejilla de botones) y el botón "comenzar" con su **ActionListener**. Cuando se presiona este botón se recogen todos los datos de los **JTextFields**, y puesto que se encuentran en forma de **Strings**, se convierten en **ints** usando el método **static Integer.parseInt( )**.

Diálogos de archivo

Algunos sistemas operativos tienen varias cajas de diálogo pre-construidas para manejar la selección de ciertos elementos como fuentes, colores, impresoras, etc. Generalmente, todos los sistemas operativos gráficos soportan la apertura y salvado de archivos, sin embargo, el **JFileChooser** de Java encapsula estas operaciones para facilitar su uso.

La aplicación siguiente ejercita dos formas de diálogos **JFileChooser**, uno para abrir y otro para guardar. La mayoría del código debería parecer ya familiar al lector, y es en los oyentes de acciones de ambos clics sobre los botones donde se dan las operaciones más interesantes:

```
//: c13:PruebaElectorArchivo.java
// Demostración de cajas de diálogo Archivo.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class PruebaElectorArchivo extends JFrame {
  JTextField
    nombreArchivo = new JTextField(),
```

```
    dir = new JTextField();
  JButton
    abrir = new JButton("Abrir"),
    salvar = new JButton("Salvar");
  public PruebaElectorArchivo() {
    JPanel p = new JPanel();
    abrir.addActionListener(new AbrirL());
    p.add(abrir);
    salvar.addActionListener(new SalvarL());
    p.add(salvar);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.SOUTH);
    dir.setEditable(false);
    nombreArchivo.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(nombreArchivo);
    p.add(dir);
    cp.add(p, BorderLayout.NORTH);
  }
  class AbrirL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Demostrar el diálogo "Abrir":
      int rVal =
        c.showOpenDialog(PruebaElectorArchivo.this);
      if(rVal == JFileChooser.APPROVE_OPTION) {
        nombreArchivo.setText(
          c.getSelectedFile().getName());
          dir.setText(
            c.getCurrentDirectory().toString());
      }
      if(rVal == JFileChooser.CANCEL_OPTION) {
        nombreArchivo.setText("Presiono Cancelar");
        dir.setText("");
      }
    }
  }
  class SalvarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Demostrar el diálogo "Salvar" :
      int rVal =
        c.showSaveDialog(PruebaElectorArchivo.this);
      if(rVal == JFileChooser.APPROVE_OPTION) {
```

```
          nombreArchivo.setText(
            c.getSelectedFile().getName());
            dir.setText(
              c.getCurrentDirectory().toString());
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
          nombreArchivo.setText("Presiono Cancelar");
          dir.setText("");
        }
      }
    }
    public static void main(String[] args) {
      Console.run(new PruebaElectorArchivo(), 250, 110);
    }
} ///:~
```

Nótese que se pueden aplicar muchas variaciones a **JFileChooser**, incluyendo filtros para limitar los nombres de archivo permitidos.

Para un diálogo abrir archivo se puede invocar a **showOpenDialog( )**, y en el caso de salvar archivo el diálogo al que se invoca es **showSaveDialog( )**. Estos comandos no devuelven nada hasta cerrar el diálogo. El objeto **JFileChooser** sigue existiendo, pero se pueden leer datos del mismo. Los métodos **getSelectedFile( )** y **getCurrentDirectory( )** son dos formas de interrogar por los resultados de la operación. Si devuelven **null** significa que el usuario canceló el diálogo.

HTML en componentes Swing

Cualquier componente que pueda tomar texto, también puede tomar texto HTML, al que dará formato de acuerdo a reglas HTML. Esto significa que se puede añadir texto elegante a los componentes Swing fácilmente. Por ejemplo:

```
/: c13:BotonHTML.java
// Poniendo texto HTML en componentes Swing.
// <applet code=BotonHTML width=200 height=500>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BotonHTML extends JApplet {
  JButton b = new JButton("<html><b><font size=+2>" +
    "<center>¡Hola!<br><i>¡Presioneme ahora!");
  public void init() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
```

```
        getContentPane().add(new JLabel("<html>"+
          "<i><font size=+4>Kapow!"));
        // Forzar la redistribución para
        // incluir la nueva etiqueta:
        validate();
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b);
  }
  public static void main(String[] args) {
    Console.run(new BotonHTML(), 200, 500);
  }
} ///:~
```

Hay que empezar el texto con "<html>" y después se pueden usar etiquetas HTML normales. Nótese que no hay obligación de incluir las etiquetas de cierre habituales.

El **ActionListener** añade una etiqueta **JLabel** nueva al formulario, que también contiene texto HTML. Sin embargo, no se añade esta etiqueta durante **init( )** por lo que hay que llamar al método **validate( )** del contenedor para forzar una redistribución de los componentes (y por consiguiente mostrar la nueva etiqueta).

También se puede usar texto HTML para **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** y **JCheckBox**.

Deslizadores y barras de progreso

Un deslizador (que ya se ha usado en el ejemplo de la onda senoidal) permite al usuario introducir datos moviéndose hacia delante y hacia atrás, lo que es intuitivo en algunas situaciones (por ejemplo, controles de volumen). Una barra de progreso muestra datos en forma relativa, desde "lleno" hasta "vacío" de forma que el usuario obtiene una perspectiva. Nuestro ejemplo favorito para éstos es simplemente vincular el deslizador a la barra de progreso de forma que al mover uno el otro varíe en consecuencia:

```
//: c13:Progreso.java
// Usando barras de progreso y deslizadoras.
// <applet code=Progreso
//  width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;
```

```
public class Progreso extends JApplet {
  JProgressBar pb = new JProgressBar();
  JSlider sb =
    new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(2,1));
    cp.add(pb);
    sb.setValue(0);
    sb.setPaintTicks(true);
    sb.setMajorTickSpacing(20);
    sb.setMinorTickSpacing(5);
    sb.setBorder(new TitledBorder("Deslizame"));
    pb.setModel(sb.getModel()); // Compartir modelo
    cp.add(sb);
  }
  public static void main(String[] args) {
    Console.run(new Progreso(), 300, 200);
  }
} ///:~
```

La clave para vincular juntos ambos elementos es compartir su modelo en la línea:

```
pb.setModel(sb.getModel());
```

Por supuesto, también se podría controlar ambos usando un oyente, pero esto es más directo en situaciones simples.

El **JProgressBar** es bastante directo, pero el **JSlider** tiene un montón de opciones, como la orientación y las marcas de mayor y menor. Nótese lo directo que es añadir un borde.

Árboles

Utilizar un **JTree** puede ser tan simple como decir:

```
add(new JTree)(
  new Object[] {"este", "ese", "aquel"}));
```

Esto muestra un árbol primitivo. Sin embargo, el API de los árboles es vasto —ciertamente uno de los mayores de Swing. Parece que casi se puede hacer cualquier cosa con los árboles, pero tareas más sofisticadas podrían requerir de bastante información y experimentación.

Afortunadamente, hay un terreno neutral en la biblioteca: los componentes árbol "por defecto", que generalmente hacen lo que se necesita. Por tanto, la mayoría de las veces se pueden usar estos componentes, y sólo hay que profundizar en los árboles en casos especiales.

El ejemplo siguiente usa los componentes árbol "por defecto" para mostrar un árbol en un *applet.* Al presionar el botón, se añade un nuevo subárbol bajo el nodo actualmente seleccionado (si no se selecciona ninguno se usa el nodo raíz):

```
//: c13:Arboles.java
// Árbol ejemplo simple de Árbol Swing. Se pueden hacer
// árboles mucho más complejos que éste.
// <applet code=Arboles
//  width=250 height=250></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.tree.*;
import com.bruceeckel.swing.*;

// Toma un array de Strings y convierte
// el primer elemento en nodo y los demás en hojas:
class Rama {
  DefaultMutableTreeNode r;
  public Rama (String[] datos) {
    r = new DefaultMutableTreeNode(data[0]);
    for(int i = 1; i < datos.length; i++)
      r.add(new DefaultMutableTreeNode(datos[i]));
  }
  public DefaultMutableTreeNode nodo() {
    return r;
  }
}

public class Arboles extends JApplet {
  String[][] datos = {
    { "Colores", "Rojo", "Azul", "Verde" },
    { "Sabores", "Acido", "Dulce", "Suave" },
    { "Longitud", "Corto", "Medio", "Largo" },
    { "Volumen", "Alto", "Medio", "Bajo" },
    { "Temperatura", "Alta", "Media", "Baja" },
    { "Intensidad", "Alta", "Media", "Baja" },
  };
  static int i = 0;
  DefaultMutableTreeNode raiz, hijo, elegido;
  JTree arbol;
  DefaultTreeModel modelo;
  public void init() {
    Container cp = getContentPane();
    raiz = new DefaultMutableTreeNode("raiz");
```

```
    arbol = new JTree(raiz);
    // Añadirlo y hacer que se encargue del desplazamiento:
    cp.add(new JScrollPane(arbol),
      BorderLayout.CENTER);
    // Capturar el modelo de árbol:
    modelo =(DefaultTreeModel)arbol.getModel();
    JButton prueba = new JButton("Pulsame");
    prueba.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(i < datos.length) {
          hijo = new rama(datos[i++]).nodo();
          // ¿Cuál es el último sobre el que se hizo clic?
          elegido = (DefaultMutableTreeNode)
            hijo.getLastSelectedPathComponent();
          if(elegido == null) elegido = raiz;
          // El modelo creara el evento
          // apropiado. En respuesta, el
          // árbol se actualizará a sí mismo:
          modelo.insertNodeInto(hijo, elegido, 0);
          // Esto pone el nuevo nodo en el nodo
          // actualmente seleccionado.
        }
      }
    });
    // Cambiar los colores de los botones:
    prueba.setBackground(Color.blue);
    prueba.setForeground(Color.white);
    JPanel p = new JPanel();
    p.add(prueba);
    cp.add(p, BorderLayout.SOUTH);
  }
  public static void main(String[] args) {
    Console.run(new Arboles(), 250, 250);
  }
} ///:~
```

La primera clase, **Rama**, es una herramienta para tomar un array de **String** y construir un **DefaultMutableTreeNode** con el primer **String** como raíz y el resto de los **String**s del array como hojas. Después se puede llamar a **nodo( )** para producir la raíz de esta "rama".

La clase **Arboles** contiene un array bi-dimensional de **String**s a partir del cual se pueden construir **Ramas**, y una **static int i** para recorrer este array. Los objetos **DefaultMutableTreeNode** guardan los nodos, pero la representación física en la pantalla la controla el **JTree** y su modelo asociado, el **DefaultTreeModel**. Nótese que cuando se añade el **JTree** al *applet*, se envuelve en un **JScrollPane** —esto es todo lo necesario para el desplazamiento automático.

El **JTree** lo controla su *modelo*. Cuando se hace un cambio al modelo, éste genera un evento que hace que el **JTree** desempeñe cualquier actualización necesaria a la representación visible del árbol. En **init( )**, se captura el modelo llamando a **getModel( )**. Cuando se presiona el botón, se crea una nueva "rama". Después, se encuentra el componente actualmente seleccionado (o se usa la raíz si es que no hay ninguno) y el método **insertNodeInto( )** del modelo hace todo el trabajo de cambiar el árbol y actualizarlo.

Un ejemplo como el de arriba puede darnos lo que necesitamos de un árbol. Sin embargo, los árboles tienen la potencia de hacer casi todo lo que se pueda imaginar —en todas partes donde se ve la expresión "por defecto" dentro del ejemplo, podría sustituirse por otra clase para obtener un comportamiento diferente. Pero hay que ser conscientes: casi todas estas clases tienen grandes interfaces, por lo que se podría invertir mucho tiempo devanándonos los sesos para entender los aspectos intrínsecos de los árboles. Independientemente de esto, constituyen un buen diseño y cualquier alternativa es generalmente mucho peor.

Tablas

Al igual que los árboles, las tablas de Swing son vastas y potentes. Su intención principal era ser la interfaz "rejilla" popular para bases de datos vía Conectividad de Bases de Datos Java (JDBC, *Java DataBase Connectivity*, que se estudiará en el Capítulo 15), y por consiguiente, tienen una gran flexibilidad, a cambio de una carga de complejidad. Aquí hay materia suficiente como para un gran rompecabezas, e incluso para justificar la escritura de un libro entero. Sin embargo, también es posible crear una **JTable** relativamente simple si se entienden las bases.

La **JTable** controla la forma de mostrar los datos, pero el **TableModel** controla los datos en sí. Por ello, para crear una tabla, se suele crear primero el **TableModel**. Se puede implementar la interfaz **TableModel** al completo pero suele ser más simple heredar de la clase ayudante **AbstractTable Model**:

```
//: c13:Tabla.java
// Demostración simple de JTable.
// <applet code=Tabla
//  width=350 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.table.*;
import javax.swing.event.*;
import com.bruceeckel.swing.*;

public class Tabla extends JApplet {
  JTextArea txt = new JTextArea(4, 20);
  // El TableModel controla todos los datos:
  class ModeloDatos extends AbstractTableModel {
    Object[][] datos = {
```

```
      {"uno", "dos", "tres", "cuatro"},
      {"cinco", "seis", "siete", "ocho"},
      {"nueve", "diez", "once", "doce"},
    };
    // Imprime los datos cuando la tabla cambia:
    class TML implements TableModelListener {
      public void tableChanged(TableModelEvent e){
        txt.setText(""); // Limpiarlo
        for(int i = 0; i < datos.length; i++) {
          for(int j = 0; j < datos[0].length; j++)
            txt.append(datos[i][j] + " ");
          txt.append("\n");
        }
      }
    }
    public ModeloDatos() {
      addTableModelListener(new TML());
    }
    public int getColumnCount() {
      return datos[0].length;
    }
    public int getRowCount() {
      return datos.length;
    }
    public Object getValueAt(int row, int col) {
      return datos[row][col];
    }
    public void
    setValueAt(Object val, int row, int col) {
      datos[row][col] = val;
      // Indica que se produjo el cambio:
      fireTableDataChanged();
    }
    public boolean
    isCellEditable(int row, int col) {
      return true;
    }
  }
  public void init() {
    Container cp = getContentPane();
    JTable tabla = new JTable(new ModeloDatos());
    cp.add(new JScrollPane(tabla));
    cp.add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
```

```
      Console.run(new Tabla(), 350, 200);
   }
} ///:~
```

ModeloDatos contiene un array de datos, pero también se podrían obtener los datos a partir de otra fuente, como una base de datos. El constructor añade un **TableModelListener** que imprime el array cada vez que se cambia la tabla. El resto de métodos siguen la convención de nombres de los Beans, y **JTable** los usa cuando quiere presentar la información en **ModeloDatos**. **AbstractTable Model** proporciona métodos por defecto para **setValueAt( )** e **isCellEditable( )** que evitan cambios a los datos, por lo que si se desea poder editar los datos, hay que superponer estos métodos.

Una vez que se tiene un **TableModel**, simplemente hay que pasárselo al constructor **JTable**. Se encargará de todos los detalles de presentación, edición y actualización. Este ejemplo también pone la **JTable** en un **JScrollPane**.

Seleccionar la Apariencia

Uno de los aspectos más interesantes de Swing es la "Apariencia conectable". Ésta permite al programa emular la apariencia y comportamiento de varios entornos operativos. Incluso se puede hacer todo tipo de cosas elegantes como cambiar dinámicamente la apariencia y el comportamiento mientras se está ejecutando el programa. Sin embargo, por lo general simplemente se desea una de las dos cosas, o seleccionar un aspecto y comportamiento "multiplataformas" (el "metal" del Swing), o seleccionar el aspecto y comportamiento del sistema en el que se está, de forma que el programa Java parece haber sido creado específicamente para ese sistema. El código para seleccionar entre estos comportamientos es bastante simple —pero hay que ejecutarlo *antes* de crear cualquier componente visual, puesto que los componentes se construirán basados en la apariencia actual y no se cambiarán simplemente porque se cambie la apariencia y el comportamiento a mitad de camino durante el programa (ese proceso es más complicado y fuera de lo común, y está relegado a libros específicos de Swing).

De hecho, si se desea usar el aspecto y comportamiento multiplataformas ("metal") característico de los programas Swing, no hay que hacer nada —es la opción por defecto. Pero si en vez de ello se desea usar el aspecto y comportamiento del entorno operativo actual, simplemente se inserta el código siguiente, generalmente el principio del **main( )** pero de cualquier forma antes de añadir componentes:

```
try {
  UIManager.setLookAndFeel(UIManager.
    getSystemLookAndFeelClassName());
} catch(Exception e) {}
```

No es necesario poner nada en la cláusula **catch** porque el **UIManager** redireccionará por defecto al aspecto y comportamiento multiplataforma si fallan los intentos de optar por otra alternativa. Sin embargo, durante la depuración, puede ser útil la excepción, pues al menos se puede desear poner una sentencia de impresión en la cláusula *catch*.

He aquí un programa que toma un parámetro de línea de comandos para seleccionar un comportamiento y una imagen, y que muestra la apariencia de varios de estos comportamientos:

```
//: c13:Apariencia.java
// Seleccionando apariencias diferentes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class Apariencia extends JFrame {
  String[] opciones = {
    "eeny", "meeny", "minie", "moe", "toe", "you"
  };
  Component[] pruebas = {
    new JButton("JButton"),
    new JTextField("JTextField"),
    new JLabel("JLabel"),
    new JCheckBox("JCheckBox"),
    new JRadioButton("Radio"),
    new JComboBox(opciones),
    new JList(opciones),
  };
  public Apariencia() {
    super("Apariencia");
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < pruebas.length; i++)
      cp.add(pruebas[i]);
  }
  private static void errorUso() {
    System.out.println(
      "Uso:Apariencia [cross|system|motif]");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length == 0) errorUso();
    if(args[0].equals("cross")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getCrossPlatformLookAndFeelClassName());
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
```

```
    } else if(args[0].equals("system")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getSystemLookAndFeelClassName());
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
    } else if(args[0].equals("motif")) {
      try {
        UIManager.setLookAndFeel("com.sun.java."+
          "swing.plaf.motif.MotifLookAndFeel");
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
    } else errorUso();
    // Nótese que la apariencia debe establecerse
    // antes de crear cualquier componente.
    Console.run(new Apariencia(), 300, 200);
  }
} ///:~
```

Se puede ver que una de las opciones es especificar explícitamente una cadena de caracteres para un aspecto y un comportamiento, como se ha visto con **MotifLookAndFeel**. Sin embargo, ése y el "metal" por defecto son los únicos que se pueden usar legalmente en cualquier plataforma; incluso aunque hay cadenas de caracteres para los aspectos y comportamientos de Windows y Macintosh, éstos sólo pueden usarse en sus plataformas respectivas (se producen al invocar a **getSystemLook AndFeelClassName( )** estando en esa plataforma particular).

También es posible crear un paquete de "apariencia" personalizado, por ejemplo, si se está construyendo un marco de trabajo para una compañía que quiere una apariencia distinta. Éste es un gran trabajo y se escapa con mucho del alcance de este libro (¡de hecho, se descubrirá que se escapa del alcance de casi todos los libros dedicados a Swing!).

El portapapeles

Las JFC soportan algunas operaciones con el portapapeles del sistema (gracias al paquete **java.awt.datatransfer**). Se pueden copiar objetos **String** al portapapeles como si de texto se tratara, y se puede pegar texto de un portapapeles dentro de objetos **String**. Por supuesto, el portapapeles está diseñado para guardar cualquier tipo de datos, pero cómo represente el portapapeles la información que en él se deposite depende de cómo haga el programa las copias y los pegados. El API de portapapeles de Java proporciona extensibilidad mediante el concepto de versión. Todos los datos provenientes del portapapeles tienen asociadas varias versiones en los que se puede convertir (por ejemplo, un gráfico podría representarse como un string de números o como una imagen) y es posible consultar si ese portapapeles en particular soporta la versión en la que uno está interesado.

El programa siguiente es una mera demostración de cortar, copiar y pegar con datos **String** dentro de un **JTextArea**. Fíjese que las secuencias de teclado que suelen usarse para cortar, copiar y pegar funcionan igualmente. Incluso si se echa un vistazo a cualquier **JTextArea** o **JTextField** de cualquier otro programa, se descubre que también soportan las secuencias de teclas correspondientes al portapapeles. Este ejemplo simplemente añade control programático del portapapeles, ofreciendo una serie de técnicas que pueden usarse siempre que se desee capturar texto en cualquier cosa que no sea un **JTextComponent**.

```
//: c13:CortarYPegar.java
// Usando el portapapeles.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import com.bruceeckel.swing.*;

public class CortarYPegar extends JFrame  {
  JMenuBar mb = new JMenuBar();
  JMenu editar = new JMenu("Editar");
  JMenuItem
    cortar = new JMenuItem("Cortar"),
    copiar = new JMenuItem("Copiar"),
    pegar = new JMenuItem("Pegar");
  JTextArea texto = new JTextArea(20, 20);
  Clipboard clipbd =
    getToolkit().getSystemClipboard();
  public CortarYPegar()  {
    cortar.addActionListener(new CortarL());
    copiar.addActionListener(new Copiar());
    pegar.addActionListener(new PegarL());
    editar.add(cortar);
    editar.add(copiar);
    editar.add(pegar);
    mb.add(editar);
    setJMenuBar(mb);
    getContentPane().add(texto);
  }
  class CopiarL implements ActionListener {
    public void actionPerformed(ActionEvent e)  {
      String seleccion = texto.getSelectedText();
      if (seleccion == null)
        return;
      StringSelection trozoCadena =
        new StringSelection(seleccion);
      clipbd.setContents(trozoCadena,trozoCadena);
```

```
      }
    }
    class CortarL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        String seleccion = texto.getSelectedText();
        if (seleccion == null)
          return;
        StringSelection trozoCadena =
          new StringSelection(seleccion);
        clipbd.setContents(trozoCadena, trozoCadena);
        texto.replaceRange("",
          texto.getSelectionStart(),
          texto.getSelectionEnd());
      }
    }
    class PegarL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        Transferable trozoDatos =
          clipbd.getContents(CortarYPegar.this);
        try {
          String trozoCadena =
            (String)trozoDatos.
              getTransferData(
                DataFlavor.stringFlavor);
          texto.replaceRange(trozoCadena,
            texto.getSelectionStart(),
            texto.getSelectionEnd());
        } catch(Exception ex) {
          System.err.println("No hay versión String");
        }
      }
    }
    public static void main(String[] args) {
      Console.run(new CortarYPegar(), 300, 200);
    }
} ///:~
```

La creación y adición del menú y del **JTextArea** deberían parecer ya una actividad de andar por casa. Lo diferente es la creación del campo **clipbd** de **Clipboard** que se lleva a cabo a través del **Toolkit**.

Toda la acción se da en los oyentes. Tanto el oyente **Copiar** como el **CortarL** son iguales excepto por la última línea de **CortarL**, que borra la línea que se está copiando. Las dos líneas especiales son la creación de un objeto **StringSelection** a partir del **String** y la llamada a **setContents( )** con esta **StringSelection**. El propósito del resto del código es poner un **String** en el portapapeles.

En **PegarL**, se extraen datos del portapapeles utilizando **getContents( )**. Lo que se obtiene es un objeto **Transferable**, bastante anónimo, y se desconoce lo que contiene. Una forma de averiguarlo es llamar a **getTransferDataFlavours( )**, que devuelve un array de objetos **DataFlavor** que indica las versiones soportadas por ese objeto en particular. También se le puede preguntar directamente con **isDataFlavorSupported( )** al que se le pasa la versión en la que uno está interesado. Aquí, sin embargo, se sigue el enfoque de invocar a **getTransferData( )** asumiendo que el contenido soporta la versión **String**, y si no lo hace se soluciona el problema en el gestor de excepciones.

Más adelante es posible que cada vez se de soporte a más versiones.

Empaquetando un applet en un archivo JAR

Un uso importante de la utilidad JAR es optimizar la carga de *applets*. En Java 1.0, la gente tendía a intentar concentrar todo su código en una única clase *applet* de forma que el cliente sólo necesitaría un acceso al servidor para descargar el código del *applet*. Esto no sólo conducía a programas complicados y difíciles de leer, sino que el archivo **.class** seguía sin estar comprimido, por lo que su descarga no era tan rápida como podría haberlo sido.

Los archivos JAR solucionan el problema comprimiendo todos los archivos **.class** en un único archivo que descarga el navegador. Ahora se puede crear el diseño correcto sin preocuparse de cuántos archivos **.class** conllevará, y el usuario obtendrá un tiempo de descarga mucho menor.

Considérese **TicTacToe.java**. Parece contener una única clase, pero, de hecho, contiene cinco clases internas, con lo que el total son seis. Una vez compilado el programa, se empaqueta en un archivo JAR con la línea:

```
jar cf TicTacToe.jar *.class
```

Ahora se puede crear una página HTML con la nueva etiqueta **archive** para indicar el nombre del archivo JAR. He aquí la etiqueta usando la forma vieja de la etiqueta HTML, a modo ilustrativo:

```
<head><tittle>TicTacToe Ejemplo Applet
</tittle></head>
<vody>
<applet code=TicTacToe.class
        archive=TicTacToe.jar
        width=200 height=100>
</applet>
</body>
```

Habrá que ponerlo en la forma nueva (más complicada y menos clara) vista anteriormente en este capítulo si queremos que funcione.

Técnicas de programación

Dado que la programación IGU en Java ha sido una tecnología en evolución con algunos cambios muy importantes entre Java 1.0/1.1 y la biblioteca Swing de Java 2, ha habido algunas estructuras y formas de obrar de programación antiguas que se han convertido precisamente en los ejemplos que vienen con Swing. Además, Swing permite programar de más y mejores formas que con los modelos viejos. En esta sección se demostrarán algunos de estos aspectos presentando y examinando algunos casos de programación.

Correspondencia dinámica de objetos

Uno de los beneficios del modelo de eventos de Swing es la flexibilidad. Se puede añadir o quitar comportamiento de eventos simplemente con llamadas a métodos. Esto se demuestra en el ejemplo siguiente:

```
//: c13:EventosDinamicos.java
// Se puede cambiar el comportamiento de los eventos dinámicamente.
// También muestra acciones múltiples para un evento.
// <applet code=EventosDinamicos
//  width=250 height=400></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class EventosDinamicos extends JApplet {
  ArrayList v = new ArrayList();
  int i = 0;
  JButton
    b1 = new JButton("Boton1"),
    b2 = new JButton("Boton2");
  JTextArea txt = new JTextArea();
  class B implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("se presiona un boton\n");
    }
  }
  class OyenteConteo implements ActionListener {
    int indice;
    public OyenteConteo(int i) { indice = i; }
    public void actionPerformed(ActionEvent e) {
      txt.append("Oyente contado "+indice+"\n");
    }
```

```
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("Boton 1 pulsado\n");
      ActionListener a = new OyenteConteo(i++);
      v.add(a);
      b2.addActionListener(a);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("Boton2 pulsado\n");
      int fin = v.size() - 1;
      if(fin >= 0) {
        b2.removeActionListener(
          (ActionListener)v.get(fin));
        v.remove(fin);
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    b1.addActionListener(new B());
    b1.addActionListener(new B1());
    b2.addActionListener(new B());
    b2.addActionListener(new B2());
    JPanel p = new JPanel();
    p.add(b1);
    p.add(b2);
    cp.add(BorderLayout.NORTH, p);
    cp.add(new JScrollPane(txt));
  }
  public static void main(String[] args) {
    Console.run(new EventosDinamicos(), 250, 400);
  }
} ///:~
```

El nuevo enfoque de este ejemplo radica en:

1. Hay más de un oyente para cada **Button**. Generalmente, los componentes manejan eventos de forma *multidifusión*, lo que quiere decir que se pueden registrar muchos oyentes para un único evento. En los componentes especiales en los que un evento se maneje de forma *unidifusion*, se obtiene una **TooManyListenersException**.

2. Durante la ejecución del programa, se añaden y eliminan dinámicamente los oyentes del **Button b2**. La adición se lleva a cabo de la forma vista anteriormente, pero cada componente también tiene un método **removeXXXListener( )** para eliminar cualquier tipo de oyente.

Este tipo de flexibilidad permite una potencia de programación muchísimo mayor.

El lector se habrá dado cuenta de que no se garantiza que se invoque a los oyentes en el orden en el que se añaden (aunque de hecho, muchas implementaciones sí que funcionan así).

Separar la lógica de negocio de la lógica IU

En general, se deseará diseñar clases de forma que cada una "sólo haga una cosa". Esto es especialmente importante cuando se ve involucrado el código de interfaz de usuario, puesto que es fácil vincular "lo que se está haciendo" con "cómo mostrarlo". Este tipo de emparejamiento evita la reutilización de código. Es mucho más deseable separar la "lógica de negocio" del IGU. De esta forma, no sólo se puede volver a usar más fácilmente la lógica de negocio, sino que también se facilita la reutilización del IGU.

Otro aspecto son los sistemas *multihilo* en los que los "objetos de negocio" suelen residir en una máquina completamente separada. Esta localización central de las reglas de negocio permite que los cambios sean efectivos al instante para toda nueva transacción, y es, por tanto, un método adecuado para actualizar un sistema. Sin embargo, se pueden usar estos objetos de negocio en muchas aplicaciones de forma que no estén vinculados a una única forma de presentación en pantalla. Es decir, su propósito debería restringirse a llevar a cabo operaciones de negocio y nada más.

El ejemplo siguiente muestra lo sencillo que resulta separar la lógica de negocio del código IGU:

```
//: c13:Separacion.java
// Separando lógica IGU de los objetos de negocio.
// <applet code=Separacion
// width=250 height=150> </applet>
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.applet.*;
import com.bruceeckel.swing.*;

class LogicaNegocio {
  private int modificador;
  public LogicaNegocio(int mod) {
    modificador = mod;
  }
  public void establecerModificador(int mod) {
    modificador = mod;
  }
```

```
  public int obtenerModificador() {
    return modificador;
  }
  // Algunas operaciones de negocio:
  public int calculo1(int arg) {
    return arg * modificador;
  }
  public int calculo2(int arg) {
    return arg + modificador;
  }
}

public class Separacion extends JApplet {
  JTextField
    t = new JTextField(15),
    mod = new JTextField(15);
  LogicaNegocio bl = new LogicaNegocio(2);
  JButton
    calc1 = new JButton("Calculo 1"),
    calc2 = new JButton("Calculo 2");
  static int obtenerValor(JTextField tf) {
    try {
      return Integer.parseInt(tf.getText());
    } catch(NumberFormatException e) {
      return 0;
    }
  }
  class Calc1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText(Integer.toString(
        bl.calculo1(obtenerValor(t))));
    }
  }
  class Calc2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText(Integer.toString(
        bl.calculo2(getValue(t))));
    }
  }
  // Si se desea que ocurra algo siempre que
  // cambia un JTextField, añade este oyente:
  class ModL implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
      bl.establecerModificador(getValue(mod));
```

```
    }
    public void removeUpdate(DocumentEvent e) {
      bl.establecerModificador(obtenerValor(mod));
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    calc1.addActionListener(new Calc1L());
    calc2.addActionListener(new Calc2L());
    JPanel p1 = new JPanel();
    p1.add(calc1);
    p1.add(calc2);
    cp.add(p1);
    mod.getDocument().
      addDocumentListener(new ModL());
    JPanel p2 = new JPanel();
    p2.add(new JLabel("Modificador:"));
    p2.add(mod);
    cp.add(p2);
  }
  public static void main(String[] args) {
    Console.run(new Separacion(), 250, 100);
  }
} ///:~
```

Se puede ver que **LogicaNegocio** es una clase directa que lleva a cabo sus operaciones sin ningún tipo de línea de código relacionada con un entorno IGU. Simplemente hace su trabajo.

Separación mantiene un seguimiento de todos los detalles de IU, y se comunica con **LogicaNegocio** sólo a través de su interfaz **public**. Todas las operaciones se centran en conseguir información de ida y vuelta a través del IU y el objeto **LogicaNegocio**. Así, a cambio **Separación** simplemente hace su trabajo. Dado que **Separación** sólo sabe que está hablando a un objeto **LogicaNegocio** (es decir, no está altamente acoplado), se podría hacer que se comunique con otros tipos de objetos sin grandes problemas.

Pensar en términos de separar IU de la lógica de negocio también facilita las cosas cuando se está adaptando código antiguo para que funcione con Java.

Una forma canónica

Las clases internas, el modelo de eventos de Swing, y el hecho de que el viejo modelo de eventos siga siendo soportado junto con las nuevas facetas de biblioteca basadas en estilos de programación antiguos, vienen a añadir elementos de confusión al proceso de diseño de código. Ahora hay incluso más medios para que la gente escriba código desagradable.

Excepto en circunstancias de extenuación, siempre se puede usar el enfoque más simple y claro: clases oyente (escritas generalmente como clases internas) para solucionar necesidades de manejo de eventos. Así se ha venido haciendo en la mayoría de ejemplos de este capítulo.

Siguiendo este modelo uno debería ser capaz de reducir las sentencias de su programa que digan: "Me pregunto qué causó este evento". Cada fragmento de código está involucrado con *hacer algo*, y no con la comprobación de tipos. Ésta es la mejor forma de escribir código; no sólo es fácil de conceptuar, sino que es aún más fácil de leer y mantener.

Programación visual y Beans

Hasta este momento, en este libro hemos visto lo valioso que es Java para crear fragmentos reusables de código. La unidad "más reusable" de código ha sido la clase, puesto que engloba una unidad cohesiva de características (campos) y comportamientos (métodos) que pueden reutilizarse directamente vía composición o mediante la herencia.

La herencia y el polimorfismo resultan partes esenciales de la programación orientada a objetos, pero en la mayoría de ocasiones, cuando se juntan en una aplicación, lo que se desea son componentes que hagan exactamente lo que uno necesita. Se desearía juntar estos fragmentos en un diseño exactamente igual que un ingeniero electrónico junta chips en una placa de circuitos. Parece también que debería existir alguna forma de acelerar este estilo de programación basado en el "ensamblaje modular".

La "programación visual" tuvo éxito por primera vez —tuvo *mucho* éxito— con el Visual Basic (VB) de Microsoft, seguido de un diseño de segunda generación en el Delphi de Borland (la inspiración primaria del diseño de los JavaBeans). Con estas herramientas de programación los componentes se representan visualmente, lo que tiene sentido puesto que suelen mostrar algún tipo de componente visual como botones o campos de texto. La representación visual, de hecho, suele ser la apariencia exacta del componente dentro del programa en ejecución. Por tanto, parte del proceso de programación visual implica arrastrar un componente desde una paleta para depositarlo en un formulario. La herramienta constructora de aplicaciones escribe el código correspondiente, y ese código será el encargado de crear todos los componentes en el programa en ejecución.

Para completar un programa, sin embargo, no basta con simplemente depositar componentes en un formulario. A menudo hay que cambiar las características de un componente, como su color, el texto que tiene, la base de datos a la que se conecta, etc. A las características que pueden modificarse en tiempo de ejecución se les denomina *propiedades*. Las propiedades de un componente pueden manipularse dentro de la herramienta constructora de aplicaciones, y al crear el programa se almacena esta información de configuración, de forma que pueda ser regenerada al comenzar el programa.

Hasta este momento, uno ya se ha acostumbrado a la idea de que un objeto es más que un conjunto de características; también es un conjunto de comportamientos. En tiempo de diseño, los comportamientos de un componente visual se representan mediante *eventos*, que indican que "aquí hay algo que le puede ocurrir al componente". Habitualmente, se decide qué es lo que se desea que ocurra cuando se da el evento, vinculando código a ese evento.

Aquí está la parte crítica: la herramienta constructora de aplicaciones utiliza la reflectividad para interrogar dinámicamente al componente y averiguar qué propiedades y eventos soporta. Una vez que sabe cuáles son, puede mostrar las propiedades y permitir cambiarlas (salvando el estado al construir el programa), y también mostrar los eventos. En general, se hace algo como doble clic en el evento y la herramienta constructora de aplicaciones crea un cuerpo de código que vincula a ese evento en particular. Todo lo que hay que hacer en ese momento es escribir el código a ejecutar cuando se dé el evento.

Todo esto conlleva a que mucho código lo haga la herramienta generadora de aplicaciones. Como resultado, uno puede centrarse en qué apariencia tiene el programa y en qué es lo que se supone que debe hacer, y confiar en la herramienta constructora de aplicaciones para que gestione los detalles de conexión. La razón del gran éxito de las herramientas constructoras de aplicaciones es que aceleran dramáticamente el proceso de escritura de una aplicación —sobretodo la interfaz de usuario, pero a menudo también otras porciones de la aplicación.

¿Qué es un Bean?

Como puede desprenderse, un Bean es simplemente un bloque de código generalmente embebido en una clase. El aspecto clave es la habilidad de la herramienta constructora de aplicaciones para descubrir las propiedades y eventos de ese componente. Para crear un componente VB, el programador tenía que escribir un fragmento de código bastante complicado siguiendo determinadas convenciones para exponer las propiedades y eventos. Delphi era una herramienta de programación visual de segunda generación y el lenguaje se diseñó activamente en torno a la programación visual, por lo que es mucho más fácil crear componentes visuales. Sin embargo, Java ha desarrollado la creación de componentes visuales hasta conducirla a su estado más avanzado con los JavaBeans, porque un Bean no es más que una clase. No hay que escribir ningún código extra o usar extensiones especiales del lenguaje para convertir algo en un Bean. Todo lo que hay que hacer, de hecho, es modificar ligeramente la forma de nombrar los métodos. Es el nombre del método el que dice a la herramienta constructora de aplicaciones si algo es una propiedad, un evento o simplemente un método ordinario.

En la documentación de Java, a esta convención de nombres se le denomina erróneamente "un patrón de diseño". Este nombre es desafortunado, puesto que los patrones de diseño (ver *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*) ya conllevan bastantes retos sin necesidad de este tipo de confusiones. No es un patrón de diseño, es simplemente una convención de nombres, y es bastante simple:

1. En el caso de una propiedad de nombre **xxx**, se suelen crear dos métodos: **getXxx( )** y **setXxx( )**. Nótese que la primera letra tras "get" y "set" se pone automáticamente en minúscula para generar el nombre de la propiedad. El tipo producido por el método "get" es el mismo que constituye el argumento para el método "set". El nombre de la propiedad y el tipo del "set" y "get" no tienen por qué guardar relación.

2. Para una propiedad **boolean**, se puede usar el enfoque "get" y "set" de arriba, pero también se puede usar "is" en vez de "get".

3. Los métodos generales del Bean no siguen la convención de nombres de arriba, siendo **public**.

4. En el caso de eventos, se usa el enfoque del "oyente" de Swing. Es exactamente lo que se ha estado viendo: **addFooBarListener(FooBarListener)** y **removeFooBarListerner(FooBarListener)** para manejar un **FooBarEvent**. La mayoría de las veces los eventos y oyentes ya definidos serán suficientes para satisfacer todas las necesidades, pero también se pueden crear interfaces oyentes y eventos propios.

El Punto 1 de los citados responde a la pregunta sobre algo que el lector podría haber visto al mirar al código viejo frente al código nuevo: muchos nombres de método han sufrido pequeños cambios de nombre, aparentemente no significativos. Ahora puede verse que la mayoría de esos cambios están relacionados con adaptar las convenciones de nombrado del "get" y "set" para convertir ese componente particular en un Bean.

Para crear un Bean simple pueden seguirse estas directrices:

```
//: beanrana:Rana.java
// Un JavaBean trivial.
package beanrana;
import java.awt.*;
import java.awt.event.*;

class Lugares {}

public class Rana {
  private int saltos;
  private Color color;
  private Lugares lugares;
  private boolean saltador;
  public int getSaltos() { return saltos; }
  public void setSaltos(int nuevosSaltos) {
    saltos = nuevosSaltos;
  }
  public Color getColor() { return color; }
  public void setColor(Color nuevoColor) {
    color = nuevoColor;
  }
  public Lugares getLugares() { return lugares; }
  public void setLugares(Lugares nuevosLugares) {
    lugares = nuevosLugares;
  }
  public boolean isSaltador() { return saltador; }
  public void setSaltador(boolean s) { saltador = s; }
  public void addActionListener(
      ActionListener l) {
```

```
    //...
  }
  public void removeActionListener(
      ActionListener l) {
    // ...
  }
  public void addKeyListener(KeyListener l) {
    // ...
  }
  public void removeKeyListener(KeyListener l) {
    // ...
  }
  // Un método público "ordinario":
  public void croar() {
    System.out.println("¡Croac!");
  }
} ///:~
```

En primer lugar, puede verse que es simplemente una clase. Generalmente todos los campos serán **private** y accesibles sólo a través de métodos. Siguiendo esta convención de nombres, las propiedades son **saltos**, **color**, **lugares** y **saltador** (nótese el cambio de mayúscula a minúscula en el caso de la primera letra). Aunque el nombre del identificador interno es el mismo que el nombre de la propiedad en los tres primeros casos, en **saltador** se puede ver que el nombre de la propiedad no obliga a usar ningún identificador particular para variables internas (ni de hecho, a *tener* ninguna variable interna para esa propiedad).

Los eventos que maneja este Bean son **ActionEvent** y **KeyEvent**, basados en el nombrado de los métodos "añadir" y "remover" del oyente asociado. Finalmente, podemos ver que el método ordinario **croar( )** sigue siendo parte del Bean simplemente por ser un método **public**, y no porque se someta a ningún esquema de nombres.

Extraer **BeanInfo** con el **Introspector**

Una de las partes más críticas del esquema de los Beans se dan al sacar un Bean de una paleta y colocarlo en un formulario. La herramienta constructora de aplicaciones debe ser capaz de crear el Bean (lo que puede hacer siempre que haya algún constructor por defecto) y después, sin acceder al código fuente del Bean, extraer toda la información necesaria para crear la hoja de propiedades y los manejadores de eventos.

Parte de la solución ya es evidente desde la parte final del Capítulo 12: la *reflectividad* de Java permite descubrir todos los métodos de una clase anónima. Esto es perfecto para solucionar el problema de los Beans sin que sea necesario usar ninguna palabra clave extra del lenguaje, como las que hay que usar en otros lenguajes de programación visual. De hecho, una de las razones primarias por las que se añadió reflectividad a Java era dar soporte a los Beans (aunque la reflectividad también soporta la serialización de objetos y la invocación remota de métodos). Por tanto, podría esperarse

que el creador de la herramienta constructora de aplicaciones hubiera aplicado reflectividad a cada Bean para recorrer sus métodos y localizar las propiedades y eventos del Bean.

Esto es verdaderamente posible, pero los diseñadores de Java querían proporcionar una herramienta estándar, no sólo para que los Beans fueran más fáciles de usar, sino también para proporcionar una pasarela estándar de cara a la creación de Beans más complejos. Esta herramienta es la clase **Introspector**, y su método más importante es el **static getBeanInfo( )**. A este método se le pasa una referencia a una **Class**, e interroga concienzudamente a la clase, devolviendo un objeto **BeanInfo** que puede ser después diseccionado para buscar en él propiedades, métodos y eventos.

Generalmente no hay que preocuparse por esto —probablemente se irán obteniendo los Beans ya diseñados por un tercero, y no habrá que conocer toda la magia subyacente en ellos. Simplemente se arrastrarán los Beans al formulario, se configurarán sus propiedades y se escribirán manipuladores para los eventos en los que se esté interesado. Sin embargo, es interesante y educativo ejercitar el uso del **Instrospector** para mostrar la información relativa a un Bean, así que he aquí una herramienta que lo hace:

```
//: c13:VolcadorBean.java
// Haciendo Introspección de un Bean.
// <applet code=VolcadorBean width=600 height=500>
// </applet>
import java.beans.*;
import java.lang.reflect.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class VolcadorBean extends JApplet {
  JTextField consulta =
    new JTextField(20);
  JTextArea resultados = new JTextArea();
  public void prt(String s) {
    resultados.append(s + "\n");
  }
  public void volcar(Class bean){
    resultados.setText("");
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(
        bean, java.lang.Object.class);
    } catch(IntrospectionException e) {
      prt("No se pudo hacer instrospeccion a " +
        bean.getName());
      return;
    }
    PropertyDescriptor[] propiedades =
```

```
      bi.getPropertyDescriptors();
    for(int i = 0; i < propiedades.length; i++) {
      Class p = propiedades[i].getPropertyType();
      prt("Tipo de la propiedad:\n  " + p.getName() +
        "Nombre de la propiedad:\n  " +
        propiedades[i].getName());
      Method leerMetodo =
        propiedades[i].getReadMethod();
      if(readMethod != null)
        prt("Leer metodo:\n  " + leerMetodo);
      Method escribirMetodo =
        propiedades[i].getWriteMethod();
      if(escribirMetodo != null)
        prt("Escribir metodo:\n  " + escribirMetodo);
      prt("====================");
    }
    prt("Metodos public:");
    MethodDescriptor[] metodos =
      bi.getMethodDescriptors();
    for(int i = 0; i < metodos.length; i++)
      prt(metodos[i].getMethod().toString());
    prt("=====================");
    prt("Soporte a eventos:");
    EventSetDescriptor[] eventos =
      bi.getEventSetDescriptors();
    for(int i = 0; i < eventos.length; i++) {
      prt("Tipo de Oyente:\n  " +
        eventos[i].getListenerType().getName());
      Method[] lm =
        eventos[i].getListenerMethods();
      for(int j = 0; j < lm.length; j++)
        prt("Metodo Oyente:\n  " +
          lm[j].getName());
      MethodDescriptor[] lmd =
        eventos[i].getListenerMethodDescriptors();
      for(int j = 0; j < lmd.length; j++)
        prt("Descriptor de metodo:\n  " +
          lmd[j].getMethod());
      Method aniadirOyente =
        eventos[i].getAddListenerMethod();
      prt("Aniadir metodo Oyente:\n  " +
          aniadirOyente);
      Method quitarOyente =
        eventos[i].getRemoveListenerMethod();
      prt("Eliminar metodo Oyente:\n  " +
```

```
            quitarOyente);
        prt("====================");
      }
    }
  class Volcador
  implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nombre = consulta.getText();
      Class c = null;
      try {
        c = Class.forName(nombre);
      } catch(ClassNotFoundException ex) {
        resultados.setText("No se pudo encontrar " + nombre);
        return;
      }
      volcar(c);
    }
  }
  public void init() {
    Container cp = getContentPane();
    JPanel p = new JPanel();
    p.setLayout(new FlowLayout());
    p.add(new JLabel("Nombre de bean cualificado:"));
    p.add(consulta);
    cp.add(BorderLayout.NORTH, p);
    cp.add(new JScrollPane(resultados));
    Volcador vlcdr = new Volcador();
    consulta.addActionListener(vlcdr);
    consulta.setText("beanRana.Rana");
    // Forzar evaluación
    vlcdr.actionPerformed(
      new ActionEvent(vlcdr, 0, ""));
  }
  public static void main(String[] args) {
    Console.run(new VolcadorBean(), 600, 500);
  }
} ///:~
```

VolcadorBean.volcar( ) es el método que hace todo el trabajo. Primero intenta crear un objeto **BeanInfo**, y si tiene éxito llama a los métodos de **BeanInfo** que producen información sobre las propiedades, métodos y eventos. En **Instrospector.getBeanInfo( )**, se verá que hay un segundo parámetro. Éste dice a **Introspector** dónde parar en la jerarquía de herencia. Aquí se para antes de analizar todos los métodos de **Object**, puesto que no nos interesan.

En el caso de las propiedades, **getPropertyDescriptors( )** devuelve un array de **PropertyDescriptors**. Por cada **PropertyDescriptor** se puede invocar a **getPropertyType( )** para averiguar la clase

del objeto pasado hacia dentro y hacia fuera vía los métodos de propiedad. Después, por cada propiedad se puede acceder a sus pseudónimos (extraídos de los métodos de nombre) con **getName( )**, el método para leer con **getReadMethod( )**, y el método para escribir con **getWriteMethod( )**. Estos dos últimos métodos devuelven un objeto **Method**, que puede, de hecho, ser usado para invocar al método correspondiente en el objeto (esto es parte de la reflectividad).

En el caso de los métodos **public** (incluyendo los métodos de propiedad), **getMethodDescriptors( )** devuelve un array de **MethodDescriptors**. Por cada uno se puede obtener el objeto **Method** asociado e imprimir su nombre.

En el caso de los eventos, **getEventSetDescriptors( )** devuelve un array de (¿qué otra cosa podría ser?) **EventSetDescriptors**. Cada uno de éstos puede ser interrogado para averiguar la clase del oyente, los métodos de esa clase oyentes, y los métodos de adición y eliminación de oyentes. El programa **VolcadorBean** imprime toda esta información.

Al empezar, el programa fuerza la evaluación de **beanrana.Rana**. La salida, una vez eliminados los detalles extra innecesarios por ahora, es:

```
class name: Rana
Tipo de la propiedad:
     Color
Nombre de la propiedad:
     color
Leer metodo:
     public Color getColor ()
Escribir metodo:
     public void setColor (Color)
====================
Tipo de la propiedad:
     lugares
Nombre de la propiedad:
     lugares
Leer metodo
     public Lugares getLugares ()
Escribir metodo:
     public void setLugares (Lugares)
====================
Tipo de la propiedad:
     boolean
Nombre de la propiedad:
     saltador
Leer metodo:
     public boolean isSaltador ()
Escribir metodo:
     public void setSaltador (boolean)
=====================
```

```
Tipo de la propiedad:
     int
Nombre de la propiedad:
     saltos
Leer metodo:
     public int getSaltos ()
Escribir metodo:
     public void setSaltos (int)
====================
Metodos public:
public void setSaltos (int)
public void croar ()
public void removeActionListener (ActionListener)
public void addActionListener (ActionListener)
public int getSaltos ()
public void setColor (Color)
public void setLugares (Lugares)
public void setSaltador (boolean)
public boolean isSaltador ()
public void addKeyListener (KeyListener)
public Color getColor ()
public void removeKeyListener (KeyListener)
public Lugares getLugares ()
====================
Soporte a Eventos:
Tipo de Oyente
     KeyOyente
Metodo Oyente:
     keyTyped
Metodo Oyente:
     keyPressed
Metodo Oyente:
     keyReleased
Descriptor de metodo:
     public void keyTyped (KeyEvent)
Descriptor de metodo:
     public void keyPressed (KeyEvent)
Descriptor de metodo:
     public void keyReleased (KeyEvent)
Añadir metodo Oyente:
     public void addKeyListener (KeyListener)
Eliminar metodo Oyente:
     public  void removeKeyListener (KeyListener)
====================
Tipo de Oyente:
```

```
        ActionListener
Tipo de Oyente:
        ActionPerformed
Descriptor de metodo:
        public void actionPerformed (ActionEvent)
Añadir metodo Oyente:
        public void addActionListener (ActionListener)
Eliminar metodo Oyente:
        public void removeActionListener (ActionListener)
====================
```

Esto revela la mayoría de lo que **Introspector** ve a medida que produce un objeto **BeanInfo** a partir del Bean. Se puede ver que el tipo de propiedad es independiente de su nombre. Nótese que los nombres de propiedad van en minúsculas. (La única ocasión en que esto no ocurre es cuando el nombre de propiedad empieza con más de una letra mayúscula en una fila.) Y recuerde que los nombres de método que se ven en este caso (como los métodos de lectura y escritura) son de hecho producidos a partir de un objeto **Method** que puede usarse para invocar al método asociado del objeto.

La lista de métodos **public** incluye la lista de métodos no asociados a una propiedad o evento, como **croar( )**, además de los que sí lo están. Éstos son todos los métodos a los que programando se puede invocar en un Bean, y la herramienta constructora de aplicaciones puede elegir listarlos todos al hacer llamadas a métodos, para facilitarte la tarea.

Finalmente, se puede ver que se analizan completamente los eventos en el oyente, sus métodos y los métodos de adición y eliminación de oyentes. Básicamente, una vez que se dispone del **BeanInfo**, se puede averiguar todo lo que sea relevante para el Bean. También se puede invocar a los métodos para ese Bean, incluso aunque no se tenga otra información excepto el objeto (otra faceta, de nuevo, de la reflectividad).

Un Bean más sofisticado

El siguiente ejemplo es ligeramente más sofisticado, a la vez que frívolo. Es un **JPanel** que dibuja un pequeño círculo en torno al ratón cada vez que se mueve el ratón. Cuando se presiona el ratón, aparece la palabra "¡Bang!" en medio de la pantalla, y se dispara un oyente de acción.

Las propiedades que pueden cambiarse son el tamaño del círculo, además del color, tamaño, y texto de la palabra que se muestra al presionar el ratón. Un **BeanExplosion** también tiene su propio **addActionListener( )** y **removeActionListener( )**, de forma que uno puede adjuntar su propio oyente a disparar cuando el usuario haga clic en el **BeanExplosion**. Habría que ser capaz de reconocer la propiedad y el soporte al evento:

```
//: beanexplosion:BeanExplosion.java
// Un Bean gráfico.
package beanexplosion;
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BeanExplosion extends JPanel
      implements Serializable {
   protected int xm, ym;
   protected int tamanioC = 20; // Tamaño del circulo
   protected String texto = "¡Bang!";
   protected int tamanioFuente = 48;
   protected Color colorT = Color.red;
   protected ActionListener oyenteAccion;
   public BeanExplosion() {
      addMouseListener(new ML());
      addMouseMotionListener(new MML());
   }
   public int getTamanioCirculo() { return tamanioC; }
   public void setTamanioCirculo(int nuevoTamanio) {
      tamanioC = nuevotamanio;
   }
   public String getTextExplosion() { return texto; }
   public void setTextExplosion(String nuevoTexto) {
      texto = nuevoTexto;
   }
   public int getTamanioFuente() { return tamanioFuente; }
   public void setTamanioFuente(int nuevoTamanio) {
      tamanioFuente = nuevoTamanio;
   }
   public Color getColorTexto() { return colorT; }
   public void setColorTexto(Color nuevoColor) {
      colorT = nuevoColor;
   }
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.setColor(Color.black);
      g.drawOval(xm - tamanioC/2, ym - tamanioC/2,
         tamanioC, tamanioC);
   }
   // Éste es un oyente unidifusión, que es la forma mas simple
   // de gestión de oyente:
   public void addActionListener (
         ActionListener l)
            throws TooManyListenersException {
```

```
      if(oyenteAccion != null)
        throw new TooManyListenersException();
      oyenteAccion = l;
    }
    public void removeActionListener(
        ActionListener l) {
      oyenteAccion = null;
    }
    class ML extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(colorT);
        g.setFont(
          new Font(
            "TimesRoman", Font.BOLD, tamanioFuente));
        int ancho =
          g.getFontMetrics().stringWidth(texto);
        g.drawString(texto,
          (getSize().width - ancho) /2,
          getSize().alto/2);
        g.dispose();
        // Llamar al método del listener:
        if(oyenteAccion != null)
          oyenteAccion.actionPerformed(
            new ActionEvent(BeanExplosion.this,
              ActionEvent.ACTION_PERFORMED, null));
      }
    }
    class MML extends MouseMotionAdapter {
      public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
      }
    }
    public Dimension getPreferredSize() {
      return new Dimension(200, 200);
    }
} ///:~
```

Lo primero que se verá es que **BeanExplosion** implementa la interfaz **Serializable**. Esto significa que la herramienta constructora de aplicaciones puede "acceder" a toda la información del **BeanExplosion** usando la serialización una vez que el diseñador del programa haya ajustado los valores de las propiedades. Cuando se crea el Bean como parte de la aplicación en ejecución se restauran estas propiedades y se realmacenan, obteniendo así exactamente lo que se diseñó.

Se puede ver que todos los campos son **private**, que es lo que generalmente se hará con un Bean —permitir el acceso sólo a través de métodos, generalmente usando el esquema "de propiedades".

Cuando se echa un vistazo a la signatura de **addActionListener( )**, se ve que puede lanzar una **TooManyListenersException**. Esto indica que es *unidifusión*, lo que significa que sólo notifica el evento a un oyente. Generalmente, se usarán eventos *multidifusión* de forma que puedan notificarse a varios oyentes. Sin embargo, eso conlleva aspectos para los que uno no estará preparado hasta acceder al siguiente capítulo, por lo que se volverá a retomar este tema en él (bajo el título de "Revisar los JavaBeans"). Un evento unidifusión "circunvala" este problema.

Cuando se hace clic con el ratón, se pone el texto en el medio del **BeanExplosion** y si el campo **actionListener** no es **null**, se invoca su **actionPerformed( )**, creando un nuevo objeto **ActionEvent** en el proceso. Cada vez que se mueva el ratón, se capturan sus nuevas coordenadas y se vuelve a dibujar el lienzo (borrando cualquier texto que esté en él, como se verá).

He aquí la clase **PruebaBeanExplosion** que permite probar el Bean como otro *applet* o aplicación:

```
//: c13:PruebaBeanExplosion.java
// <applet code=PruebaBeanExplosion
// width=400 height=500></applet>
import bangbean.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class PruebaBeanExplosion extends JApplet {
  JTextField txt = new JTextField(20);
  // Durante la prueba, informar de las acciones:
  class BBL implements ActionListener {
    int conteo = 0;
    public void actionPerformed(ActionEvent e){
      txt.setText("Accion BeanExplosion "+ conteo++);
    }
  }
  public void init() {
    BeanExplosion bb = new BeanExplosion();
    try {
      bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
      txt.setText("Demasiados oyentes");
    }
    Container cp = getContentPane();
    cp.add(bb);
    cp.add(BorderLayout.SOUTH, txt);
```

```
  }
  public static void main(String[] args) {
    Console.run(new PruebaBeanExplosion(), 400, 500);
  }
} ///:~
```

Cuando un Bean está en un entorno de desarrollo, no se usará su clase, pero es útil proporcionar un método de prueba rápida para cada uno de los Beans. **PruebaBeanExplosion** coloca un **BeanExplosion** dentro del *applet*, adjuntando un **ActionListener** simple al **BeanExplosion** para imprimir una cuenta de eventos al **JTextField** cada vez que se da un **ActionEvent**. Generalmente, por supuesto, la herramienta constructora de aplicaciones crearía la mayoría de código que usa el Bean.

Cuando se ejecuta el **BeanExplosion** a través de **VolcadoBean** o se pone **BeanExplosion** dentro de un entorno de desarrollo que soporta Beans, se verá que hay muchas más propiedades y acciones de lo que parece en el código de arriba. Esto es porque **BeanExplosion** se hereda de **JPanel**, y **JPanel** es también un Bean, por lo que se están viendo también sus propiedades y eventos.

Empaquetar un Bean

Antes de poder incorporar un Bean en una herramienta de construcción visual habilitada para Beans, hay que ponerlo en un contenedor estándar de Beans, que es un archivo JAR que incluye todas las clases Bean además de un archivo de *manifiesto* que dice: "Esto es un Bean". Un archivo *manifiesto* no es más que un archivo de texto que sigue un formato particular. En el caso del **BeanExplosion**, el archivo *manifiesto* tiene la apariencia siguiente (sin la primera y última líneas):

```
//:! :BeanExplosion.mb
Manifest-Version: 1.0

Name: BeanExplosion/BeanExplosion.class
Java-Bean: True
///:~
```

La primera línea indica la versión del esquema de *manifiesto*, que mientras Sun no indique lo contrario es la 1.0. La segunda línea (las líneas en blanco se ignoran) nombra el archivo **BeanExplosion.class**, y la tercera dice, "Es un Bean". Sin la tercera línea, la herramienta constructora de programas no reconocería la clase como un Bean.

El único punto del que hay que asegurarse es de que se pone el nombre correcto en el campo "Name". Si se vuelve atrás a **BeanExplosion.java**, se verá que está en el **package beanexplosion** (y por consiguiente en un subdirectorio denominado "beanexplosion" que está fuera del *classpath*), y el nombre del archivo de *manifiesto* debe incluir esta información de paquete. Además, hay que colocar el archivo de *manifiesto* en el directorio *superior* a la raíz de la trayectoria de los paquetes, lo que en este caso significa ubicar el archivo en el directorio padre al subdirectorio "beanexplosion". Posteriormente hay que invocar a **jar** desde el directorio que contiene al archivo de *manifiesto*, así:

```
jar cfm BeanExplosion.jar BeanExplosion.mf beanExplosion
```

Esta línea asume que se desea que el archivo JAR resultante se denomine **BeanExplosion.jar**, y que se ha puesto el *manifiesto* en un archivo denominado **BeanExplosion.mf**.

Uno podría preguntarse "¿Qué ocurre con todas las demás clases que se generaron al compilar **BeanExplosion.java**?" Bien, todas acabaron dentro del subdirectorio **beanexplosion**, y se verá que el último parámetro para la línea de comandos de **jar** de arriba es el subdirectorio **beanexplosión**. Cuando se proporciona a **jar** el nombre de un subdirectorio, empaqueta ese subdirectorio al completo en el archivo jar (incluyendo, en este caso, el archivo de código fuente **BeanExplosion.java** original —uno podría elegir no incluir el código fuente con sus Beans). Además, si se deshace esta operación y se desempaqueta el archivo JAR creado, se descubrirá que el archivo de *manifiesto* no está dentro, pero que **jar** ha creado su propio archivo de *manifiesto* (basado parcialmente en el nuestro) de nombre **MANIFEST.MF** y lo ha ubicado en el subdirectorio **META-INF** (es decir, "metainformación"). Si se abre este archivo de *manifiesto* también se verá que **jar** ha añadido información de firma digital por cada archivo, de la forma:

```
Digest-Algorithm: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=
MD5-Digest: O4NcS1hE3Smnzlp2hj6qeg==
```

En general, no hay que preocuparse por nada de esto, y si se hacen cambios se puede modificar simplemente el archivo de *manifiesto* original y volver a invocar a **jar** para que cree un archivo JAR nuevo para el Bean. También se pueden añadir otros Beans al archivo JAR simplemente añadiendo su información al de *manifiesto*.

Una cosa en la que hay que fijarse es que probablemente se querrá poner cada Bean en su propio subdirectorio, puesto que cuando se crea un archivo JAR se pasa a la utilidad **jar** el nombre de un subdirectorio y éste introduce todo lo que hay en ese subdirectorio en el archivo JAR. Se puede ver que tanto **Rana** como **BeanExplosion** están en sus propios subdirectorios.

Una vez que se tiene el Bean adecuadamente insertado en el archivo JAR, se puede incorporar a un entorno constructor de programas habilitado para Beans. La manera de hacer eso cambia de una herramienta a otra, pero Sun proporciona un banco de pruebas para JavaBeans disponible gratuitamente en su *Beans Development Kit* (BDK) denominado el "beanbox". (El BDK puede descargarse de *http://java.sun.com/beans*.) Para ubicar un Bean en la beanbox, se copia el archivo JAR en el directorio "jars" del BDK antes de iniciar el beanbox.

Soporte a Beans más complejo

Se puede ver lo remarcadamente simple que resulta construir un Bean. Pero uno no está limitado a lo que ha visto aquí. La arquitectura de JavaBeans proporciona un punto de entrada simple, pero también se puede escalar a situaciones más complejas. Estas situaciones van más allá del alcance de este libro, pero se presentarán de forma breve en este momento. Hay más detalles en *http://java.sun.com/beans*.

Un lugar en el que se puede añadir sofisticación es con las propiedades. Los ejemplos de arriba sólo mostraban propiedades simples, pero también es posible representar múltiples propiedades en un array. A esto se le llama una *propiedad indexada*. Simplemente se proporcionan los métodos ade-

cuados (siguiendo de nuevo una convención para los nombres de los métodos) y el **Introspector** reconoce la propiedad indexada de forma que la herramienta constructora de aplicaciones pueda responder de forma apropiada.

Las propiedades pueden *vincularse*, lo que significa que se notificarán a otros objetos vía un **PropertyChangeEvent**. Los otros objetos pueden después elegir cambiarse a sí mismos basándose en el cambio sufrido por el Bean.

Las propiedades pueden *limitarse*, lo que significa que los otros objetos pueden vetar cambios en esa propiedad si no los aceptan. A los otros objetos se les avisa usando un **PropertyChangeEvent**, y puede lanzar una **PropertyVetoException** para evitar que ocurra el cambio y restaurar los valores viejos.

También se puede cambiar la forma de representar el Bean en tiempo de diseño:

1. Se puede proporcionar una hoja general de propiedades del Bean en particular. Esta hoja de propiedades se usará para todos los otros Beans, invocándose el nuestro automáticamente al seleccionar el Bean.

2. Se puede crear un editor personalizado para una propiedad particular, de forma que se use la hoja de propiedades ordinaria, pero cuando se edite la propiedad especial, se invocará automáticamente a este editor.

3. Se puede proporcionar una clase **BeanInfo** personalizada para el Bean, que produzca información diferente de la información por defecto que crea en **Introspector**.

4. También es posible pasar a modo "experto" o no por cada **FeatureDescriptor**s para distinguir entre facetas básicas y aquéllas más complicadas.

Más sobre Beans

Hay otro aspecto que no se pudo describir aquí. Siempre que se cree un Bean, cabría esperar que se ejecute en un entorno multihilo. Esto significa que hay que entender los conceptos relacionados con los hilos, y que se presentarán en el Capítulo 14. En éste se verá una sección denominada "Volver a visitar los Beans" que describirá este problema y su solución.

Hay muchos libros sobre JavaBeans; por ejemplo, *JavaBeans* de Elliotte Rusty Harold (IDG, 1998).

Resumen

De todas las bibliotecas de Java, la biblioteca IGU ha visto los cambios más dramáticos de Java 1.0 a Java 2. La AWT de Java 1.0 se criticó como uno de los peores diseños jamás vistos, y aunque permitía crear programas portables, el IGU resultante era "igualmente mediocre en todas las plataformas". También imponía limitaciones y era extraño y desagradable de usar en comparación con las herramientas de desarrollo de aplicaciones nativas disponibles en cada plataforma particular.

Cuando Java 1.1 presentó el nuevo modelo de eventos y los JavaBeans, se dispuso el escenario —de forma que era posible crear componentes IGU que podían ser arrastrados y depositados fácilmente dentro de herramientas de construcción de aplicaciones visuales. Además, el diseño del modelo de eventos y de los Beans muestra claramente un alto grado de consideración por la facilidad de programación y la mantenibilidad del código (algo que no era evidente en la AWT de Java 1.0). Pero hasta que no aparecieron las clases JFC/Swing, el trabajo no se dio por concluido. Con los componentes Swing, la programación de IGU multiplataforma se convirtió en una experiencia civilizada.

De hecho, lo único que se echa de menos es la herramienta constructora de aplicaciones, que es donde radica la verdadera revolución. El Visual Basic y Visual C++ de Microsoft requieren de herramientas de construcción de aplicaciones propias de Microsoft, al igual que ocurre con Borland en el caso de Delphi y C++. Si se desea que la herramienta de construcción de aplicaciones sea mejor, hay que cruzar los dedos y esperar a que el productor haga lo que uno espera. Pero Java es un entorno abierto, y por tanto, no sólo permite entornos de construcción de aplicaciones de otros fabricantes, sino que trata de hacer énfasis en que se construyan. Y para que estas herramientas sean tomadas en serio, deben soportar JavaBeans. Esto significa un campo de juego por niveles: si sale una herramienta constructora de aplicaciones mejor, uno no está obligado a usar la que venía usando —puede decidir cambiarse a la nueva e incrementar su productividad. Este tipo de entorno competitivo para herramientas constructoras de aplicaciones IGU no se había visto nunca anteriormente, y el mercado resultante no puede sino ser beneficioso para la productividad del programador.

Este capítulo solamente pretendía dar una introducción a la potencia de Swing al lector de forma que pudiera ver lo relativamente simple que es introducirse en las bibliotecas. Lo visto hasta la fecha será probablemente suficiente para una porción significativa de necesidades de diseño de IU. Sin embargo, Swing tiene muchas más cosas —se pretende que sea un conjunto de herramientas de extremada potencia para el diseño de IU. Probablemente proporciona alguna forma de lograr absolutamente todo lo que se nos pudiera ocurrir.

Si uno no ve aquí todo lo que necesita, siempre puede optar por la documentación en línea de Sun y buscar en la Web, y si sigue siendo poco, buscar un libro dedicado a Swing —un buen punto de partida es *The JFC Swing Tutorial*, de Walrath & Campione (Addison Welsey, 1999).

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com.*

1. Crear un *applet*/aplicación usando la clase **Console** mostrada en este capítulo. Incluir un campo de texto y tres botones. Al presionar cada botón, hacer que aparezca algún texto distinto en el campo de texto.

2. Añadir una casilla de verificación al *applet* creado en el Ejercicio 1, capturar el evento e insertar un texto distinto en el campo de texto.

3. Crear un *applet*/aplicación utilizando **Console**. En la documentación HTML de *java.sun.com*, encontrar el **JPasswordField** y añadirlo al programa. Si el usuario teclea la contraseña correcta, usar **JOptionPane** para proporcionar un mensaje de éxito al usuario.

4. Crear un *applet*/aplicación usando **Console**, y añadir todos los componentes que tengan un método **addActionListener( )**. (Consultarlos todos en la documentación HTML de *http://java.sun.com*. Truco: usar el índice.) Capturar sus eventos y mostrar un mensaje apropiado para cada uno dentro de un campo de texto.

5. Crear un *applet*/aplicación usando **Console**, con un **JButton**, y un **JTextField**. Escribir y adjuntar el oyente apropiado de forma que si el foco reside en el botón, los caracteres del mismo aparezcan en el **JTextField**.

6. Crear un *applet*/aplicación usando **Console**. Añadir al marco principal todos los componentes descritos en este capítulo, incluyendo menús y una caja de diálogo.

7. Modificar **CamposTexto.java** de forma que los caracteres de **t2** retengan el uso de mayúsculas y minúsculas original, en vez de forzarlas automáticamente a mayúsculas.

8. Localizar y descargar uno o más entornos de desarrollo constructores de IGU gratuitos disponibles en Internet, o comprar un producto comercial. Descubrir qué hay que hacer para añadir **BeanExplosion** al entorno, y hacerlo.

9. Añadir **Rana.class** al archivo de *manifiesto* como se muestra en este capítulo y ejecutar **jar** para crear un archivo JAR que contenga tanto a **Rana** como a **BeanExplosion**. Ahora descargar e instalar el BDK de Sun, o bien usar otra herramienta constructora de programas habilitada para Beans, y añadir el archivo JAR al entorno, de forma que se puedan probar estos dos Beans.

10. Crear su propio JavaBean denominado **Valvula**, que contenga dos propiedades: una **boolean** denominada "on" y otra **int** denominada "nivel". Crear un archivo de *manifiesto*, usar **jar** para empaquetar el Bean, y después cargarlo en la beanbox o en una herramienta constructora de programas habilitada para Beans, y así poder probarlo.

11. Modificar **CajasMensajes.java** de forma que tenga un **ActionListener** individual por cada botón (en vez de casar el texto del botón).

12. Monitorizar un nuevo tipo de evento en **RastrearEvento.java**, añadiendo el nuevo código de manejo del evento. Primero habrá que decidir cuál será el nuevo tipo de evento a monitorizar.

13. Heredar un nuevo tipo de botón de **JButton**. Cada vez que se presione este botón, debería cambiar de color a un valor seleccionado al azar. Ver **CajasColores.java** del Capítulo 14 para tener un ejemplo de cómo generar un valor de código al azar.

14. Modificar **PanelTexto.java** para que use un **JTextArea** en vez de un **JTextPane**.

15. Modificar **Menus.java** de forma que use botones de opción en vez de casillas de verificación en los menús.

16. Simplificar **Lista.java** pasándole el array al constructor y eliminando la adición dinámica de elementos a la lisa.

17. Modificar **OndaSeno.java** para que **DibujarSeno** sea un JavaBean añadiendo métodos "getter" y "setter".

18. ¿Recuerdas el juguete "Telesketch", con dos controles, uno encargado del movimiento vertical del punto del dibujo, y otro que controla el movimiento horizontal? Crear uno de ésos usando **OndaSeno.java** como comienzo. En vez de controles rotatorios, usar barras de desplazamiento. Añadir un botón que borre toda la pantalla.

19. Crear un "indicador de progresos asintótico" que vaya cada vez más despacio a medida que se acerca a su meta. Añadir comportamiento errático al azar de forma que periódicamente parezca que comienza a acelerar.

20. Modificar **Progreso.java** de forma que no comparta modelos sino que use un oyente para conectar el deslizador y la barra de progreso.

21. Seguir las instrucciones de la sección "Empaquetando un *applet* en un archivo JAR" para ubicar **TicTacToe.java** en un archivo JAR. Crear una página HTML con la versión (complicada y difícil) de la etiqueta *applet*, y modificarla para que use la etiqueta de archivo para usar el archivo JAR. (Truco: empezar con la página HTML de **TicTacToe.java** que viene con la distribución de código fuente de este libro.)

22. Crear un *applet*/aplicación usando **Console**. Este debería tener tres deslizadores, para los valores del rojo, verde y azul de **java.awt.Color**. El resto del formulario debería ser un **JPanel** que muestre el color determinado por los tres deslizadores. Incluir también campos de texto no editables que muestren los valores RGB actuales.

23. En la documentación HTML de **javax.swing**, buscar **JColorChooser**. Escribir un programa con un botón que presente este selector de color como un diálogo.

24. Casi todo componente Swing se deriva de **Component**, que tiene un método **setCursor( )**. Buscarlo en la documentación HTML. Crear un *applet* y cambiar el cursor a uno de los almacenados en la clase **Cursor**.

25. A partir de **MostrarAddListeners.java**, crear un programa con la funcionalidad completa de **LimpiarMostrarMetodos.java** del Capítulo 12.

14: Hilos múltiples

Los objetos proporcionan una forma de dividir un programa en secciones independientes. A menudo, también es necesario convertir un programa en subtareas separadas que se ejecuten independientemente.

Cada una de estas subtareas independientes recibe el nombre de *hilo*, y uno programa como si cada hilo se ejecutara por sí mismo y tuviera la UCP para él sólo. Algún mecanismo subyacente divide de hecho el tiempo de UCP entre ellos, pero generalmente el programador no tiene por qué pensar en ello, lo que hace de la programación de hilos múltiples una tarea mucho más sencilla.

Un *proceso* es un programa en ejecución autocontenido con su propio espacio de direcciones. Un sistema operativo *multitarea* es capaz de ejecutar más de un proceso (programa) a la vez, mientras hace que parezca como si cada uno fuera el único que se está ejecutando, proporcionándole ciclos de UCP periódicamente. Por consiguiente, un único proceso puede tener múltiples hilos ejecutándose concurrentemente.

Hay muchos usos posibles del multihilo, pero, en general, se tendrá parte del programa vinculado a un evento o recurso particular, no deseando que el resto del programa pueda verse afectado por esta vinculación. Por tanto, se crea un hilo asociado a ese evento o tarea y se deja que se ejecute independientemente del programa principal. Un buen ejemplo es un botón de "salir" —no hay por qué verse obligado a probar el botón de salir en todos los fragmentos de código que se escriban en el programa, aunque sí se desea que el botón de salir responda, como si se *estuviera comprobando* regularmente. De hecho, una de las razones más importantes para la existencia del multihilo es la existencia de interfaces de usuario que respondan rápidamente.

Interfaces de respuesta de usuario rápida

Como punto de partida, puede considerarse un programa que lleva a cabo alguna operación intensa de UCP y que acaba ignorando la entrada de usuario y por tanto no emite respuestas. Éste, un applet/aplicación, simplemente mostrará el resultado de un contador en ejecución:

```
//: c14:Contador1.java
// Una interfaz de usuario que no responde.
// <applet code=Contador1 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Contador1 extends JApplet {
```

```
  private int conteo = 0;
  private JButton
    empezar = new JButton("Empezar"),
    onOff = new JButton("Conmutar");
  private JTextField t = new JTextField(10);
  private boolean flagEjecutar = true;
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    empezar.addActionListener(new EmpezarL());
    cp.add(empezar);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public void comenzar() {
    while (true) {
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
      if (FlagEjecutar)
        t.setText(Integer.toString(count++));
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      comenzar();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      flagEjecutar = !flagEjecutar;
    }
  }
  public static void main(String[] args) {
    Console.run(new Contador1(), 300, 100);
  }
} ///:~
```

En este punto, el código del *applet* y Swing deberían ser racionalmente familiares, al estar explicados en el Capítulo 13. En el método **comenzar( )** es donde permanece ocupado el programa: pone el valor actual de **conteo** en el **JTextField t**, y después incrementa **conteo**.

Parte del bucle infinito interno a **comenzar( )** llama a **sleep( )**. Éste debe estar asociado con un objeto **Thread**, y resulta que toda aplicación tiene *algún* hilo asociado a él. (De hecho, Java se basa en hilos y siempre hay alguna ejecución junto con la aplicación.) Por tanto, independientemente de si se usan o no hilos de forma explícita, se puede producir el hilo actual que usa el programa con **Hilos** y el método **static sleep( )**.

Nótese que **sleep( )** puede lanzar una **InterruptedException**, aunque lanzar esta excepción se considera una forma hostil de romper un hilo, por lo que no se recomienda. (Una vez más, las excepciones son para condiciones excepcionales, no el flujo normal de control.) La capacidad de interrumpir a un hilo durmiente se ha incluido para soportar una faceta futura del lenguaje.

Cuando se presiona el botón **Empezar**, se invoca a **comenzar( )**. Al examinar **comenzar( )**, se podría pensar estúpidamente (como hicimos) que debería permitir el multihilo porque se va a dormir. Es decir, mientras que el método está dormido, parece como si la UCP pudiera estar ocupada monitorizando otras presiones sobre el botón. Pero resulta que el problema real es que **comenzar( )** nunca devuelve nada, puesto que está en un bucle infinito, y esto significa que **actionPerformed( )** nunca devuelve nada. Puesto que uno está enclavado en **actionPerformed( )** debido a la primera vez que se presionó el botón, el programa no puede gestionar ningún otro evento. (Para salir, de alguna forma hay que matar el proceso; la forma más sencilla de hacerlo es presionar Control-C en la ventana de la consola, si es que se lanzó desde la consola. Si se empieza vía el navegador, hay que matar la ventana del navegador.)

El problema básico aquí es que **comenzar( )** necesita continuar llevando a cabo sus operaciones, y al mismo tiempo necesita devolver algo, de forma que **actionPerformed( )** pueda completar su operación y la interfaz de usuario continúe respondiendo al usuario. Pero en un método convencional como **comenzar( )** no puede continuar *y* al mismo tiempo devolver el control al resto del programa. Esto suena a imposible de lograr, como si la UCP debiera estar en dos lugares a la vez, pero ésta es precisamente la ilusión que proporciona el multihilo.

El modelo de hilos (y su soporte de programación en Java) es una conveniencia de programación para simplificar estos juegos malabares y operaciones que se dan simultáneamente en un único programa. Con los hilos, la UCP puede ir rotando y dar a cada hilo parte de su tiempo. Cada hilo tiene impresión de tener la UCP para sí mismo durante todo el tiempo. La excepción se da siempre que el programa se ejecute en múltiples UCP. Pero uno de los aspectos más importantes de los hilos es que uno se abstrae de esta capa, de forma que el código no tiene por qué saber si se está ejecutando en una o en varias UCP. Por consiguiente, los hilos son una forma de crear programas transparentemente escalables.

Los hilos pueden reducir algo la eficiencia de computación, pero la mejora en el diseño de programa, el balanceo de recursos y la conveniencia del usuario suelen ser muy valiosos. Por supuesto, si se tiene más de una UCP, el sistema operativo puede dedicar cada UCP a un conjunto de hilos o incluso a un único hilo, logrando que el programa, en su globalidad, se ejecute mucho más rápido. La multitarea y el multihilo tienden a ser las formas más razonables de usar sistemas multiprocesador.

Heredar de **Thread**

La forma más simple de crear un hilo es heredar de la clase **Thread**, que tiene todo lo necesario para crear y ejecutar hilos. El método más importante de **Thread** es **run( )**, que debe ser sobreescrito para hacer que el hilo haga lo que se le mande. Por consiguiente, **run( )** es el código que se ejecutará "simultáneamente" con los otros hilos del programa.

El ejemplo siguiente crea cualquier número de hilos de los que realiza un seguimiento asignando a cada uno con un único número, generado con una variable **static**. El método **run( )** de **Thread** se sobreescribe para que disminuya cada vez que pase por el bucle y acabe cuando valga cero (en el momento en que acabe **run( )**, se termina el hilo).

```
//: c14:HiloSimple.java
// Ejemplo muy simple de hilos.

public class HiloSimple extends Thread {
  private int cuentaAtras = 5;
  private static int conteoHilos = 0;
  private int numeroHilo = ++conteoHilos;
  public HiloSimple() {
    System.out.println("Creando " + numeroHilo);
  }
  public void run() {
    while(true) {
      System.out.println("Hilo " +
        numeroHilo + "(" + cuentaAtras + ")");
      if(--cuentaAtras == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new HiloSimple().start();
    System.out.println("Todos los hilos Arrancados");
  }
} ///:~
```

Un método **run( )** suele tener siempre algún tipo de bucle que continúa hasta que el hilo deja de ser necesario, por lo que hay que establecer la condición en la que romper el bucle y salir (o, en el caso de arriba, simplemente **return** de **run( )**). A menudo, se convierte **run( )** en forma de bucle infinito, lo que significa que a falta de algún factor externo que haga que **run( )** termine, continuará para siempre.

En el método **main( )** se puede ver el número de hilos que se están creando y ejecutando. El método **start( )** de la clase **Thread** lleva a cabo alguna inicialización especial para el hilo y después llama a **run( )**. Por tanto, los pasos son: se llama al constructor para que contruya el objeto, después

start( ) configura el hilo y llama a **run( )**. Si no se llama a **start( )** (lo que puede hacerse en el constructor si es apropiado) nunca se dará comienzo al hilo.

La salida de una ejecución de este programa (que será distinta cada vez) es:

```
Creando 1
Creando 2
Creando 3
Creando 4
Creando 5
Hilo 1(5)
Hilo 1(4)
Hilo 1(3)
Hilo 1(2)
Hilo 2(5)
Hilo 2(4)
Hilo 2(3)
Hilo 2(2)
Hilo 2(1)
Hilo 1(1)
Todos los hilos Arrancados
Hilo 3(5)
Hilo 4(5)
Hilo 4(4)
Hilo 4(3)
Hilo 4(2)
Hilo 4(1)
Hilo 5(5)
Hilo 5(4)
Hilo 5(3)
Hilo 5(2)
Hilo 5(1)
Hilo 3(4)
Hilo 3(3)
Hilo 3(2)
Hilo 3(1)
```

Se verá que en este ejemplo no se llama nunca a **sleep( )**, y la salida sigue indicando que cada hilo obtiene una porción del tiempo de UCP en el que ejecutarse. Esto muestra que **sleep( )**, aunque descansa en la existencia de un hilo para poder ejecutarse, no está involucrado en la habilitación o deshabilitación de hilos. Es simplemente otro método.

También puede verse que los hilos no se ejecutan en el orden en el que se crean. De hecho, el orden en que la UCP atiende a un conjunto de hilos existente es indeterminado, a menos que se cambien las prioridades haciendo uso del método **setPriority( )** de **Thread**.

Cuando **main( )** crea los objetos **Thread** no captura las referencias a ninguno de ellos. Un objeto ordinario debería ser un juego justo para la recolección de basura, pero no un **Thread**. Cada **Thread** "se registra" a sí mismo de forma que haya una referencia al mismo en algún lugar y el recolector de basura no pueda limpiarlo.

Hilos para una interfaz con respuesta rápida

Ahora es posible solucionar el problema de **Contador1.java** con un hilo. El truco es colocar la subtarea —es decir, el bucle de dentro de **comenzar( )**— dentro del método **run( )** de un hilo. Cuando el usuario presione el botón **empezar**, se arranca el hilo, pero después se completa la *creación* del hilo, por lo que aunque se esté ejecutando el hilo, puede continuar el trabajo principal del programa (estando pendiente y respondiendo a eventos de la interfaz de usuario). He aquí la solución:

```
//: c14:Contador2.java
// Una interfaz de usuario de respuesta rápida con hilos.
// <applet code=Contador2 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Contador2 extends JApplet {
  private class SubtareaSeparada extends Thread {
    private int conteo = 0;
    private boolean flagEjecutar = true;
    SubtareaSeparada() { start(); }
    void invertirflag() { flagEjecutar = !flagEjecutar; }
    public void run() {
      while (true) {
       try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
       if(flagEjecutar)
         t.setText(Integer.toString(conteo++));
      }
    }
  }
  private SubtareaSeparada sp = null;
  private JTextField t = new JTextField(10);
  private JButton
    empezar = new JButton("Empezar"),
    onOff = new JButton("Conmutar");
```

```
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp == null)
        sp = new SubtareaSeparada();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp != null)
        sp.invertirFlag();
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    empezar.addActionListener(new StartL());
    cp.add(empezar);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public static void main(String[] args) {
    Console.run(new Contador2 (), 300, 100);
  }
} ///:~
```

Contador2 es un programa directo, cuya única tarea es establecer y mantener la interfaz de usuario. Pero ahora, cuando el usuario presiona el botón **empezar**, el código de gestión de eventos no llama a un método. En su lugar, se crea un hilo de clase **SubTareaSeparada**, y continúa el bucle de eventos de **Counter2**.

La clase **SubTareaSeparada** es una simple extensión de **Thread** con un constructor que ejecuta el hilo invocando a **start( )**, y un método **run( )** que esencialmente contiene el código "**comenzar( )**" de **Contador1.java**.

Dado que **SubtareaSeparada** es una clase interna, puede acceder directamente al **JTextField t** de **Contador2**; se puede ver que esto ocurre dentro de **run( )**. El campo **t** de la clase externa es **private** puesto que **SubTareaSeparada** puede acceder a él sin ningún permiso especial —y siempre es bueno hacer campos "tan **private** como sea posible", de forma que puedan ser cambiados accidentalmente por fuerzas externas a la clase.

Cuando se presiona el botón **onOff** conmuta el **flagEjecutar** de dentro del objeto **SubTareaSeparada**. Ese hilo (cuando mira al *flag*) puede empezar y pararse por sí mismo. Presionar el botón **onOff** produce una respuesta aparentemente instantánea. Por supuesto, la respuesta no es verdaderamente instantánea, no como la de un sistema dirigido por interrupciones. El contador sólo se detiene cuando el hilo tiene la UCP y se da cuenta de que el *flag* ha cambiado.

Se puede ver que la clase interna **SubTareaSeparada** es **private**, lo que significa que sus campos y métodos pueden tener el acceso por defecto (excepto en el caso de **run( )**, que debe ser **public**, puesto que es **public** en la clase base). La clase **private** interna no está accesible más que a **Contador2**, y ambas clases están fuertemente acopladas. En cualquier momento en que dos clases parezcan estar fuertemente acopladas entre sí, hay que considerar las mejoras de codificación y mantenimiento que se obtendrían utilizando clases internas.

Combinar el hilo con la clase principal

En el ejemplo de arriba puede verse que la clase hilo está separada de la clase principal del programa. Esto tiene mucho sentido y es relativamente fácil de entender. Sin embargo, hay una forma alternativa que se verá a menudo que no está tan clara, pero que suele ser más concisa (y que es probablemente lo que la dota de popularidad). Esta forma combina la clase principal del programa con la clase hilo haciendo que la clase principal del programa sea un hilo. Puesto que para un programa IGU la clase principal del programa debe heredarse de **Frame** o de **Applet**, hay que usar una interfaz para añadirle funcionalidad adicional. A esta interfaz se le denomina **Runnable**, y contiene el mismo método básico que **Thread**. De hecho, **Thread** también implementa **Runnable**, lo que sólo especifica la existencia de un método **run( )**.

El *uso* de programa/hilo combinado no es tan obvio. Cuando empieza el programa se crea un objeto que es **Runnable**, pero no se arranca el hilo. Esto hay que hacerlo explícitamente. Esto se puede ver en el programa siguiente, que reproduce la funcionalidad de **Contador2**:

```
//: c14:Contador3.java
// Usando la interfaz Runnable para convertir la clase
// principal en un hilo.
// <applet code=Contador3 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Contador3
    extends JApplet implements Runnable {
  private int conteo = 0;
  private boolean flagEjecutar = true;
  private Thread hiloPropio = null;
  private JButton
    empezar = new JButton("Empezar"),
    onOff = new JButton("Conmutar");
  private JTextField t = new JTextField(10);
  public void run() {
    while (true) {
      try {
```

```
        hiloPropio.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
      if(flagEjecutar)
        t.setText(Integer.toString(conteo++));
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(hiloPropio == null) {
        hiloPropio = new Thread(Contador3.this);
        hiloPropio.start();
      }
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      flagEjecutar = !flagEjecutar;
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    empezar.addActionListener(new EmpezarL());
    cp.add(empezar);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public static void main(String[] args) {
    Console.run(new Contador3(), 300, 100);
  }
} ///:~
```

Ahora el **run( )** está dentro de la clase, pero sigue estando dormido tras completarse **init( )**. Cuando se presiona el botón **Empezar**, se crea el hilo (si no existe ya) en una expresión bastante oscura:

```
new Thread(Contador3.this)
```

Cuando algo tiene una interfaz **Runnable**, simplemente significa que tiene un método **run( )**, pero no hay nada especial en ello —no produce ninguna habilidad innata a los hilos, como las de una clase heredada de **Thread**. Por tanto, para producir un hilo a partir de un objeto **Runnable**, hay que crear un objeto **Thread** separado como se mostró arriba, pasándole el objeto **Runnable** al constructor **Thread** especial. Después se puede llamar al **start( )** de ese hilo:

```
hiloPropio.start();
```

Esta sentencia lleva a cabo la inicialización habitual y después llama a **run( )**.

El aspecto conveniente de la **interface Runnable** es que todo pertenece a la misma clase. Si es necesario acceder a algo, simplemente se hace sin recorrer un objeto separado. Sin embargo, como se vio en el capítulo anterior, este acceso es tan sencillo como usar una clase interna[1].

Construir muchos hilos

Considérese la creación de muchos hilos distintos. Esto no se puede hacer con el ejemplo de antes, por lo que hay que volver hacia atrás, cuando se tenían clases separadas heredadas de **Thread** para encapsular el método **run( )**. Pero ésta es una solución más general y más fácil de entender, por lo que mientras que el ejemplo anterior muestra un estilo de codificación muy abundante, no podemos recomendarlo para la mayoría de los casos porque es un poco más confuso y menos flexible.

El ejemplo siguiente repite la forma de los ejemplos de arriba con contadores y botones de conmutación. Pero ahora toda la información de un contador particular, incluyendo el botón y el campo de texto, están dentro de su propio objeto, heredado de **Thread**. Todos los campos de **Teletipo** son **private**, lo que significa que se puede cambiar la implementación de **Teletipo** cuando sea necesario, incluyendo la cantidad y tipo de componentes de datos a adquirir y la información a mostrar. Cuando se crea un objeto **Teletipo**, el constructor añade sus componentes visuales al panel contenedor del objeto externo:

```
//: c14:Contador4.java
// Manteniendo el hilo como una clase distinta,
// se puede tener tantos hilos como se desee.
// <applet code=Contador4 width=200 height=600>
// <param name=tamanio value="12"></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Contador4 extends JApplet {
  private JButton empezar = new JButton("Empezar");
  private boolean empezado = false;
  private Teletipo[] s;
  private boolean esApplet = true;
  private int tamanio = 12;
  class Teletipo extends Thread {
    private JButton b = new JButton("Conmutar");
```

[1] **Runnable** ya estaba en Java 1.0, mientras que las clases internas no se introdujeron hasta Java 1.1, que pueda deberse probablemente a la existencia de **Runnable**. También las arquitecturas multihilo tradicionales se centraron en que se ejecutara una función en vez de un objeto. Preferimos heredar de **Thread** siempre que se pueda; nos parece más claro y más flexible.

```
    private JTextField t = new JTextField(10);
    private int conteo = 0;
    private boolean flagEjecutar = true;
    public Teletipo() {
      b.addActionListener(new Conmutador());
      JPanel p = new JPanel();
      p.add(t);
      p.add(b);
      // Invoca JApplet.getContentPane().add():
      getContentPane().add(p);
    }
    class ConmutadorL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        flagEjecutar = !flagEjecutar;
      }
    }
    public void run() {
      while (true) {
        if (flagEjecutar)
          t.setText(Integer.toString(conteo++));
        try {
          sleep(100);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!empezado) {
        empezado = true;
        for (int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Capturar el parámetro "tamanio" de la página a tamanio Web:
    if (esApplet) {
      String tam = getParameter("tamanio");
      if(tam != null)
        tamanio = Integer.parseInt(tam);
```

```
      }
      s = new Teletipo[tamanio];
      for (int i = 0; i < s.length; i++)
         s[i] = new Teletipo();
      empezar.addActionListener(new EmpezarL());
      cp.add(empezar);
   }
   public static void main(String[] args) {
      Contador4 applet = new Contador4();
      // Esto no es un applet, por lo que se pone el flag a uno
      // y se producen los valores de parámetros para args:
      applet.esApplet = false;
      if(args.length != 0)
         applet.size = Integer.parseInt(args[0]);
      Console.run(applet, 200, applet.tamanio * 50);
   }
} ///:~
```

Teletipo no sólo contiene su equipamiento como hilo, sino que también incluye la forma de controlar y mostrar el hilo. Se pueden crear tantos hilos como se desee sin crear explícitamente los componentes de ventanas.

En **Contador4** hay un array de objetos **Teletipo** llamado **s**. Para maximizar la flexibilidad, se inicializa el tamaño de este array saliendo a la página web utilizando los parámetros del applet. Esto es lo que aparenta el parámetro tamanio en la página, insertado en la etiqueta applet:

```
<param name=tamanio value="20">
```

Las palabras clave **param**, **name** y **value** pertenecen a HTML. La palabra **name** es aquello a lo que se hará referencia en el programa, y **value** puede ser una cadena de caracteres, no sólo algo que desemboca en un número.

Se verá que la determinación del tamaño del array **s** se hace dentro de **init( )**, y no como parte de una definición de **s**. Es decir, *no se puede* decir como parte de la definición de clase (fuera de todo método):

```
int tamanio = Integer.parseInt(getParameter("Tamanio"));
Teletipo[] s = new Teletipo[tamanio];
```

Esto se puede compilar, pero se obtiene una "null-pointer exception" extraña en tiempo de ejecución. Funciona bien si se mueve la inicialización **getParameter( )** dentro de **init( )**. El marco de trabajo *applet* lleva a cabo la inicialización necesaria en los parámetros antes de **init( )**.

Además, este código puede ser tanto un *applet* como una aplicación. Cuando es una aplicación, se extrae el parámetro **tamanio** de la línea de comandos (o se utiliza un valor por defecto).

Una vez que se establece el tamaño del array, se crean nuevos objetos **Teletipo**; como parte del constructor **Teletipo** se añade al *applet* el botón y el campo texto de cada **Teletipo**.

Presionar el botón **empezar** implica recorrer todo el array de **Teletipos** y llamar al método **start( )** de cada uno. Recuérdese que **start( )** lleva a cabo la inicialización necesaria por cada hilo, invocando al método **run( )** del hilo.

El oyente **ConmutadorL** simplemente invierte el *flag* de **Teletipo**, de forma que cuando el hilo asociado tome nota, pueda reaccionar de forma acorde.

Uno de los aspectos más valiosos de este ejemplo es que permite crear fácilmente conjuntos grandes de subtareas independientes además de monitorizar su comportamiento. En este caso, se verá que a medida que crece el número de tareas, la máquina mostrará mayor divergencia en los números que muestra debido a la forma de servir esos hilos.

También se puede experimentar para descubrir la importancia de **sleep(100)** dentro de **teletipo.run( )**. Si se retira el **sleep( )**, todo funcionará correctamente hasta presionar un botón de conmutar. Después, ese hilo particular tendrá un **flagEjecutar** falso, y el **run( )** se verá envuelto en un bucle infinito y rígido, que parece difícil de romper, haciendo que el grado de respuesta y la velocidad del programa descienda drásticamente.

Hilos demonio

Un hilo "demonio" es aquél que supuestamente proporciona un servicio general en segundo plano mientras se está ejecutando el programa, no siendo parte de la esencia del programa. Por consiguiente, cuando todos los hilos no demonio acaban, se finaliza el programa. Consecuentemente, mientras se siga ejecutando algún hilo no demonio, el programa no acabará. (Por ejemplo, puede haber un hilo ejecutando el método **main( )**.)

Se puede averiguar si un hilo es un demonio llamando a **isDaemon( )**, y se puede activar o desactivar el funcionamiento como demonio de un hilo con **setDaemon( )**. Si un hilo es un demonio, todos los hilos que cree serán a su vez demonios.

El ejemplo siguiente, demuestra los hilos demonio:

```
//: c14:Demonios.java
// Comportamiento endemoniado.
import java.io.*;

class Demonio extends Thread {
  private static final int TAMANIO = 10;
  private Thread[] t = new Thread[TAMANIO];
  public Demonio() {
    setDaemon(true);
    start();
  }
  public void run() {
    for(int i = 0; i < TAMANIO; i++)
      t[i] = new EngendrarDemonio(i);
```

```
    for(int i = 0; i < TAMANIO; i++)
        t [i] = new EngendrarDemonio(i);
        for (int = 0; <TAMANIO; i++)
      System.out.println(
        "t[" + i + "].isDaemon() = "
        + t[i].isDaemon());
    while(true)
      yield();
  }
}

class EngendrarDemonio extends Thread {
  public EngendrarDemonio(int i) {
    System.out.println(
      "EngendrarDemonio " + i + " empezado");
    start();
  }
  public void run() {
    while(true)
      yield();
  }
}

public class Demonios {
  public static void main(String[] args)
  throws IOException {
    Thread d = new Demonio();
    System.out.println(
      "d.isDaemon() = " + d.isDaemon());
    // Permitir a los hilos demonio
    // acabar sus procesos de arranque:
    System.out.println("Presione cualquier tecla");
    System.in.read();
  }
} ///:~
```

El hilo **Demonio** pone su hilo a true y después engendra otros muchos hilos para mostrar que son también demonios. Después se introduce en un bucle infinito y llama a **yield( )** para ceder el control a los otros procesos. En una versión anterior de este programa, los bucles infinitos incrementarían contadores **int**, pero eso parecía bloquear todo el programa. Usar **yield( )** hace que el ejemplo sea bastante picante.

No hay nada para evitar que el programa termine una vez que acabe el método **main( )**, puesto que no hay nada más que hilos demonio en ejecución. Para poder ver los resultados de lanzar todos los hilos demonio se coloca **System.in** para leer, de forma que el programa espere una pulsación de tecla antes de terminar. Sin esto sólo se ve alguno de los resultados de la creación de los hilos demonio. (Puede probarse a reemplazar el código de **read( )** con llamadas a **sleep( )** de varias longitudes y observar el comportamiento.)

Compartir recursos limitados

Se puede pensar que un programa de un hilo es una entidad solitaria que recorre el espacio del problema haciendo sólo una cosa en cada momento. Dado que sólo hay una entidad, no hay que pensar nunca que pueda haber dos entidades intentando usar el mismo recurso a la vez, como si fueran dos conductores intentando aparcar en el mismo sitio, o atravesar la misma puerta simultáneamente, o incluso, hablar.

Con la capacidad multihilo, los elementos dejan de ser solitarios, y ahora existe la posibilidad de que dos o más hilos traten de usar el mismo recurso limitado a la vez. Hay que prevenir las colisiones por un recurso o, de lo contrario, se tendrán dos hilos intentando acceder a la misma cuenta bancaria a la vez, o imprimir en la misma impresora o variar la misma válvula, etc.

Acceder a los recursos de forma inadecuada

Considérese una variación de los contadores que se han venido usando hasta ahora en el capítulo. En el ejemplo siguiente, cada hilo contiene dos contadores que se incrementan y muestran dentro de **run( )**. Además, hay otro hilo de clase **Observador** que vigila los contadores para que siempre sean equivalentes. Ésta parece una actividad innecesaria, puesto que mirando al código parece obvio que los contadores siempre tendrán el mismo valor. Pero es justamente ahí donde aparece la sorpresa. He aquí la primera versión del programa:

```
//: c14:Compartiendo1.java
// Problemas con la compartición de recursos y los hilos.
// <applet code=Compartiendo1 width=350 height=500>
// <param name=tamanio value="12">
// <param name=observadores value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Compartiendo1 extends JApplet {
  private static int recuentoAcceso = 0;
  private static JTextField conteoA =
    new JTextField("0", 7);
  public static void incrementarAcceso() {
    recuentoAcceso++;
    conteoA.setText(Integer.toString(recuentoAcceso));
  }
  private JButton
    empezar = new JButton("Empezar"),
    observador = new JButton("Vigilar");
  private boolean esApplet = true;
```

```
private int numContadores = 12;
private int numObservadores = 15;
private DosContadores[] s;
class DosContadores extends Thread {
  private boolean empezado = false;
  private JTextField
    t1 = new JTextField(5),
    t2 = new JTextField(5);
  private JLabel l =
    new JLabel("conteo1 == conteo2");
  private int conteo1 = 0, conteo2 = 0;
  // Añadir los componentes a mostrar como un panel:
  public DosContadores() {
    JPanel p = new JPanel();
    p.add(t1);
    p.add(t2);
    p.add(l);
    getContentPane().add(p);
  }
  public void start() {
    if(!empezado) {
      empezado = true;
      super.start();
    }
  }
  public void run() {
    while (true) {
      t1.setText(Integer.toString(conteo1++));
      t2.setText(Integer.toString(conteo2++));
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
  public void pruebaSinc() {
    Compartiendo1.incrementarAcceso();
    if(conteo1 != conteo2)
      l.setText("Sin sincronizar");
  }
}
class Observador extends Thread {
  public Observador() { start(); }
  public void run() {
```

```
      while(true) {
        for(int i = 0; i < s.length; i++)
          s[i].pruebaSinc();
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class ObservadorL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numObservadores; i++)
        new Observador();
    }
  }
  public void init() {
    if(esApplet) {
      String contadores = getParameter("tamanio");
      if(contadores != null)
        numContadores = Integer.parseInt(contadores);
      String observadores = getParameter("observadores");
      if(observadores != null)
        numObservadores = Integer.parseInt(observadores);
    }
    s = new DosContadores[numContadores];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new DosContadores();
    JPanel p = new JPanel();
    empezar.addActionListener(new EmpezarL());
    p.add(empezar);
    observador.addActionListener(new ObservadorL());
    p.add(observador);
    p.add(new JLabel("Cuenta de accesos"));
    p.add(conteoA);
    cp.add(p);
```

```
    }
    public static void main(String[] args) {
      Compartiendo1 applet = new Compartiendo1();
      // Esto no es un applet, por lo que se pone el flag a uno
      // y se producen los valores de parámetros para args:
      applet.esApplet = false;
      applet.numContadores =
        (args.length == 0 ? 12 :
          Integer.parseInt(args[0]));
      applet.numObservadores =
        (args.length < 2 ? 15 :
          Integer.parseInt(args[1]));
      Console.run(applet, 350,
        applet.numObservadores * 50);
    }
} ///:~
```

Como antes, cada contador sólo contiene sus componentes propios a visualizar: dos campos de texto y una etiqueta que inicialmente indica que los contadores son equivalentes. Estos componentes se añaden al panel de contenidos del objeto de la clase externa en el constructor **DosContadores**.

Dado que el hilo **DosContadores** empieza vía una pulsación de tecla por parte del usuario, es posible que se llame a **start( )** más de una vez. Es ilegal que se llame a **Thread.start( )** más de una vez para un mismo hilo (se lanza una excepción). Se puede ver la maquinaria que evita esto en el flag **empezado** y el método **start( )** superpuesto.

En **run( )**, se incrementan y muestran **conteo1** y **conteo2,** de forma que parecen ser idénticos. Después se llama a **sleep( )**; sin esta llamada el programa se detiene bruscamente porque la UCP tiene dificultad para conmutar las tareas.

El método **pruebaSinc( )** lleva a cabo la aparentemente inútil actividad de comprobar si **conteo1** es equivalente a **conteo2**; si no son equivalentes, pone la etiqueta a "Sin Sincronizar" para indicar esta circunstancia. Pero primero llama a un miembro estático de la clase **Compartiendo1,** que incrementa y muestra un contador de accesos para mostrar cuántas veces se ha dado esta comprobación con éxito. (La razón de esto se hará evidente en variaciones ulteriores de este ejemplo.)

La clase **Observador** es un hilo cuyo trabajo es invocar a **pruebaSinc( )** para todos los objetos de **DosContenedores** activos. Lo hace recorriendo el array mantenido en el objeto **Compartiendo1**. Se puede pensar que **Observador** está mirando constantemente por encima del hombro de los objetos **DosContadores**.

Compartiendo1 contiene un array de objetos **DosContenedores** que inicializa en **init( )** y comienza como hilos al presionar el botón "empezar". Más adelante, al presionar el botón "Vigilar", se crean uno o más vigilantes que se liberan sobre los hilos **DosContadores**.

Nótese que para ejecutar esto como un *applet* en un navegador, la etiqueta *applet* tendrá que contener las líneas:

```
<param name=tamanio value="20">
```

```
<param name=observadores value="1">
```

Se puede experimentar variando la anchura, altura y parámetros para satisfacer los gustos de cada uno. Cambiando el **tamaño** y los **observadores**, se puede variar el comportamiento del programa. Este programa está diseñado para ejecutarse como una aplicación independiente a la que se pasan los parámetros por la línea de comandos (o proporcionando valores por defecto).

He aquí la parte más sorprendente. En **DosContadores.run( )**, se va pasando repetidamente por el bucle infinito recorriendo las líneas siguientes:

```
t1.setText(Integer.toString(conteo1++));
t2.setText(Integer.toString(conteo2++));
```

(además de dormirse, pero eso ahora no importa). Al ejecutar el programa, sin embargo, se descubrirá que se observarán **conteo1** y **conteo2** (por parte de los **Observadores**) ¡para que a veces no sean iguales! Esto se debe a la naturaleza de los hilos —que pueden ser suspendidos en cualquier momento. Por ello, en ocasiones, se da la suspensión *justo* cuando se ha ejecutado la primera de estas líneas y no la segunda, y aparece el hilo **Observador** ejecutando la comprobación justo en ese momento, descubriendo, por consiguiente, que ambos hilos son distintos.

Este ejemplo muestra un problema fundamental del uso de los hilos. Nunca se sabe cuándo se podría ejecutar un hilo. Imagínese sentado en una mesa con un tenedor, justo a punto de engullir el último fragmento de comida del plato y justo cuando el tenedor va a alcanzarla, la comida simplemente se desvanece (porque se suspendió el hilo y apareció otro que robó la comida). Éste es el problema con el que se está tratando.

En ocasiones, no importa que un mismo recurso esté siendo accedido a la vez que se está intentado usar (la comida está en algún otro plato). Pero para que el multihilo funcione, es necesario disponer de alguna forma de evitar que dos hilos accedan al mismo recurso, al menos durante ciertos periodos críticos.

Evitar este tipo de colisión es simplemente un problema de poner un bloqueo en el recurso cuando lo esté usando un hilo. El primer hilo que accede al recurso lo bloquea, de forma que posteriormente los demás hilos no pueden acceder a este recurso hasta que éste quede desbloqueado, momento en el que es otro el hilo que lo bloquea y usa, etc. Si el asiento delantero de un coche es un recurso limitado, el primer niño que grite: "¡Para mí!", lo habrá bloqueado.

Cómo comparte Java los recursos

Java tiene soporte integrado para prevenir colisiones sobre cierto tipo de recurso: la memoria de un objeto. Puesto que generalmente se hacen los elementos de datos de clase **private** y se accede a esa memoria sólo a través de métodos, se pueden evitar las colisiones haciendo que un método particular sea **synchronized**. Sólo un hilo puede invocar a un método **synchronized** en cada instante para cada objeto (aunque ese hilo puede invocar a más de un método **synchronized** de varios objetos). He aquí métodos **synchronized** sencillos:

```
synchronized void f()  { /* ... */ }
```

```
synchronized void g() { /* ... */ }
```

Cada objeto contiene un único bloqueo (llamado también *monitor*) que forma parte del objeto automáticamente (no hay que escribir ningún código especial). Cuando se llama a cualquier método **synchronized**, se bloquea el objeto y no se puede invocar a ningún otro método **synchronized** del objeto hasta que el primero acabe y libere el bloqueo. En el ejemplo de arriba, si se invoca a **f( )** de un objeto, no se puede invocar a **g( )** de ese mismo objeto hasta que se complete **f( )** y libere el bloqueo. Por consiguiente, hay un único bloqueo que es compartido por todos los métodos **synchronized** de un objeto en particular, y este bloqueo evita que la memoria en común sea escrita por más de un método en cada instante (es decir, más de un hilo en cada momento).

También hay un único bloqueo por clase (como parte del objeto **Class** de la clase), de forma que los métodos **synchronized static** pueden bloquearse mutuamente por accesos simultáneos a datos **static** en el ámbito de una clase.

Nótese que si se desea proteger algún recurso de accesos simultáneos por parte de múltiples hilos, se puede hacer forzando el acceso a ese recurso mediante métodos **synchronized**.

Sincronizar los contadores

Armado con esta nueva palabra clave, parece que la solución está a mano: simplemente se usará la palabra **synchronized** para los métodos de **DosContadores**. El ejemplo siguiente es igual que el anterior, con la adición de la nueva palabra:

```
//: c14:Compartiendo2.java
// Usando la palabra synchronized para evitar
// el acceso múltiple a un recurso en particular.
// <applet code=Compartiendo2 width=350 height=500>
// <param name=tamanio value="12">
// <param name=observadores value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Compartiendo2 extends JApplet {
  DosContadores[] s;
  private static int recuentoAcceso = 0;
  private static JTextField conteoA =
    new JTextField("0", 7);
  public static void incrementarAccesos() {
    recuentoAcceso++;
    conteoA.setText(Integer.toString(recuentoAcceso));
  }
  private JButton
    empezar = new JButton("Empezar"),
```

```
    observador = new JButton("Vigilar");
  private boolean esApplet = true;
  private int numContadores = 12;
  private int numObservadores = 15;

  class DosContadores extends Thread {
    private boolean empezado = false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("conteo1 == conteo2");
    private int conteo1 = 0, conteo2 = 0;
    public DosContadores() {
      JPanel p = new JPanel();
      p.add(t1);
      p.add(t2);
      p.add(l);
      getContentPane().add(p);
    }
    public void start() {
      if(!empezado) {
        empezado = true;
        super.start();
      }
    }
    public synchronized void run() {
      while (true) {
        t1.setText(Integer.toString(conteo1++));
        t2.setText(Integer.toString(conteo2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
    public synchronized void pruebaSinc() {
      Compartiendo2.incrementarAcceso();
      if(conteo1 != conteo2)
        l.setText("Sin Sincronizar");
    }
  }

  class Observador extends Thread {
```

```
    public Observador() { start(); }
    public void run() {
      while(true) {
        for(int i = 0; i < s.length; i++)
          s[i].pruebaSinc();
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class ObservadorL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numObservadores; i++)
        new Observador();
    }
  }
  public void init() {
    if(esApplet) {
      String contadores = getParameter("tamanio");
      if(contadores != null)
        numContadores = Integer.parseInt(contadores);
      String observadores = getParameter("observadores");
      if(observadores != null)
        numObservadores = Integer.parseInt(observadores);
    }
    s = new DosContadores[numContadores];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new DosContadores();
    JPanel p = new JPanel();
    empezar.addActionListener(new EmpezarL());
    p.add(empezar);
    observador.addActionListener(new ObservadorL());
    p.add(observador);
    p.add(new Label("Cuenta de accesos"));
```

```
      p.add(ConteoA);
      cp.add(p);
   }
   public static void main(String[] args) {
      Compartiendo2 applet = new Compartiendo2();
      // Esto no es un applet, por lo que se pone el flag a uno
      // y se producen los valores de parámetros para args:
      applet.esApplet = false;
      applet.numContadores =
         (args.length == 0 ? 12 :
            Integer.parseInt(args[0]));
      applet.numObservadores =
         (args.length < 2 ? 15 :
            Integer.parseInt(args[1]));
      Console.run(applet, 350,
         applet.numContadores * 50);
   }
} ///:~
```

Se verá que, *tanto* **run( )** *como* **pruebaSinc( )** son **synchronized**. Si se sincroniza sólo uno de los métodos, el otro es libre de ignorar el bloqueo del objeto y accederlo con impunidad. Éste es un punto importante: todo método que acceda a recursos críticos compartidos debe ser **synchronized** o no funcionará correctamente.

Ahora aparece un nuevo aspecto. El **Observador** nunca puede saber qué está ocurriendo exactamente porque todo el método **run( )** está **synchronized**, y dado que **run( )** siempre se está ejecutando para cada objeto, el bloqueo siempre está activado y no se puede llamar nunca a **pruebaSinc ( )**. Esto se puede ver porque **RecuentoAcceso** nunca cambia.

Lo que nos gustaría de este ejemplo es alguna forma de aislar sólo *parte* del código dentro de **run( )**. La sección de código que se desea aislar así se denomina una *sección crítica* y la palabra clave **synchronized** se usa de forma distinta para establecer una sección crítica. Java soporta secciones críticas con el *bloque sincronizado*; esta vez, **synchronized** se usa para especificar el objeto cuyo bloqueo se usa para sincronizar el código adjunto:

```
synchronized(objetoSinc) {
   // A este código sólo puede
   // acceder un thread a la vez
}
```

Antes de poder entrar al bloque sincronizado, hay que adquirir el bloqueo en **objetoSinc**. Si algún otro hilo ya tiene este bloqueo, no se puede entrar en este bloque hasta que el bloqueo ceda.

El ejemplo **Compartiendo2** puede modificarse quitando la palabra clave **synchronized** de todo el método **run( )** y pendiendo en su lugar un bloque **synchronized** en torno a las dos líneas críticas. Pero ¿qué objeto debería usarse como bloqueo? El que ya está involucrado en **pruebaSinc( )**, que es el objeto actual ¡(**this**)! Por tanto, el método **run( )** modificado es así:

```
public void run() {
   while (true) {
      synchronized(this) {
         t1.setText(Integer.toString(conteo1++));
         t2.setText(Integer.toString(conteo2++));
      }
      try {
         sleep(500);
      } catch(InterruptedException e) {
         System.err.println("Interrumpido");
      }
   }
}
```

Éste es el único cambio que habría que hacer a **Compartiendo2.java**, y se verá que, mientras que los dos contadores nunca están fuera de sincronismo (de acuerdo al momento en que **Observador** puede consultar su valor), se sigue proporcionando un acceso adecuado a **Observador** durante la ejecución de **run( )**.

Por supuesto, toda la sincronización depende de la diligencia del programador: todo fragmento de código que pueda acceder a un recurso compartido deberá envolverse en un bloque sincronizado.

Eficiencia sincronizada

Dado que tener dos métodos que escriben al mismo fragmento de información no parece *nunca* ser una buena idea, podría parecer que tiene sentido que todos los métodos sean automáticamente **synchronized** y eliminar de golpe todas las palabras **synchronized**. (Por supuesto, el ejemplo con un **synchronized run( )** muestra que esto tampoco funcionaría.) Pero resulta que adquirir un bloqueo no es una operación barata —multiplica el coste de una llamada a un método (es decir, la entrada y salida del método, no la ejecución del método) al menos por cuatro, y podría ser mucho más dependiente de la implementación en sí. Por tanto, si se sabe que un método en particular no causará problemas de contención, es mejor no poner la palabra clave **synchronized**. Por otro lado, dejar de lado la palabra **synchronized** por considerarla un cuello de botella, esperando que no se den colisiones, es una invitación al desastre.

Revisar los JavaBeans

Ahora que se entiende la sincronización, se puede echar un nuevo vistazo a los JavaBeans. Cuando se cree un Bean, hay que asumir que se ejecutará en un entorno multihilo. Esto significa que:

1. Siempre que sea posible, todos los métodos **public** de un Bean deberían ser **synchronized**. Por supuesto, esto incurre en cierta sobrecarga en tiempo de ejecución. Si eso es un problema, se pueden dejar no **synchronized** los métodos que no causen problemas en secciones críticas, pero hay que tener en cuenta que esto no siempre es obvio. Los métodos encargados de calificar suelen ser pequeños (como es el caso de **getTamanioCirculo()** en el ejemplo siguiente) y/o "atómicos", es decir, la llamada al método se ejecuta en una cantidad de código

tan pequeña que el objeto no puede variar durante la ejecución. Hacer estos métodos no **synchronized** podría no tener un efecto significativo en la velocidad de ejecución de un programa. También se podrían hacer **public** todos los métodos de un Bean. También se podrían hacer **synchronized** todos los métodos **public** de un Bean, y eliminar la palabra clave **synchronized** sólo cuando se tiene la total seguridad de que es necesario hacerlo, y que su eliminación surtirá algún efecto.

2. Al disparar un evento multidifusión a un conjunto de oyentes interesados, hay que asumir que se podrían añadir o eliminar oyentes al recorrer la lista.

Es bastante fácil operar con el primer punto, pero el segundo requiere pensar un poco más. Considérese el ejemplo **BeanExplosion.java** del capítulo anterior. Éste eludía el problema del multihilo ignorando la palabra clave **synchronized** (que no se había presentado aún) y haciendo el evento unidifusión. He aquí un ejemplo modificado para que funcione en un evento multihilo y use la "multidifusión" para los eventos:

```
//: c14:BeanExplosion2.java
// Habría que escribir los Beans así para que puedan
// ejecutarse en un entorno multihilo.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class BeanExplosion2 extends JPanel
    implements Serializable {
  private int xm, ym;
  private int tamanioC = 20; // Tamaño del círculo
  private String texto = "¡Bang!";
  private int tamanioFuente = 48;
  private Color colorT = Color.red;
  private ArrayList OyentesAccion =
    new ArrayList();
  public BeanExplosion2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
  public synchronized int getTamanioCirculo () {
    return tamanioC;
  }
  public synchronized void
  setTamanioCirculo(int nuevoTamanio) {
    tamanioC = nuevoTamanio;
  }
```

```
  public synchronized String getTextoExplosion() {
    return texto;
  }
  public synchronized void
  setTextoExplosion(String nuevoTexto) {
    texto = nuevoTexto;
  }
  public synchronized int getTamanioFuente() {
    return tamanioFuente;
  }
  public synchronized void
  setTamanioFuente(int nuevoTamanio) {
    tamanioFuente = nevoTamanio;
  }
  public synchronized Color getColorTexto() {
    return colorT;
  }
  public synchronized void
  setColorTexto(Color nuevoColor) {
    colorT = nuevoColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    g.drawOval(xm - tamanioC/2, ym - tamanioC/2,
      tamanioC, tamanioC);
  }
  // Éste es el oyente multidifusión, que suele usarse
  // más a menudo que la aproximación
  // unidifusion realizada en BeanExplosion.java:
  public synchronized void
    addActionListener(ActionListener l) {
    oyentesAccion.add(l);
  }
  public synchronized void
    removeActionListener(ActionListener l) {
    oyentesAccion.remove(l);
  }
  // Nótese que esto no es sincronizado:
  public void notifyListeners() {
    ActionEvent a =
      new ActionEvent(BeanExplosion2.this,
        ActionEvent.ACTION_PERFORMED, null);
    ArrayList lv = null;
    // Hacer una copia superficial de la Lista en el caso
```

```
      // de que alguien añada un oyente mientras estamos
      // invocando a oyentes:
      synchronized(this) {
        lv = (ArrayList)oyentesAccion.clone();
      }
      // Llamar a los métodos del oyente:
      for(int i = 0; i < lv.size(); i++)
        ((ActionListener)lv.get(i))
          .actionPerformed(a);
    }
    class ML extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(ColorT);
        g.setFont(
          new Font(
            "TimesRoman", Font.BOLD, tamanioFuente));
        int ancho =
          g.getFontMetrics().stringWidth(texto);
        g.drawString(texto,
          (getSize().width - ancho) /2,
          getSize().height/2);
        g.dispose();
        notifyListeners();
      }
    }
    class MM extends MouseMotionAdapter {
      public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
      }
    }
    public static void main(String[] args) {
      BeanExplosion2 bb = new BeanExplosion2();
      bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          System.out.println("ActionEvent" + e);
        }
      });
      bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          System.out.println("Accion BeanExplosion2 ");
        }
      });
```

```
      bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          System.out.println("Mas accion");
        }
      });
      Console.run(bb, 300, 300);
    }
} ///:~
```

Añadir **synchronized** a los métodos es un cambio sencillo. Sin embargo, hay que darse cuenta de que en **addActionListener( )** y **removeActionListener( )** se añaden y eliminan **ActionListeners** de un **ArrayList**, por lo que se puede tener tantos como se quiera.

Se puede ver que el método **notifyListeners( )** *no* es **synchronized**. Puede ser invocado desde más de un hilo simultáneamente. También es posible que **addActionListener( )** o **removeActionListener( )** sean invocados en el medio de una llamada a **notifyListeners( )**, lo que supone un problema puesto que recorre el **ArrayList oyentesAccion**. Para aliviar el problema, se clona el **ArrayList** dentro de una cláusula **synchronized** y se recorre el clon (véase Apéndice A para obtener más detalles sobre la clonación). De esta forma se puede manipular el **ArrayList** original sin que esto suponga ningún impacto sobre **notifyListeners( )**.

El método **paintComponent( )** tampoco es **synchronized**. Decidir si sincronizar o no métodos superpuestos no está tan claro como al añadir métodos propios. En este ejemplo, resulta que **paint( )** parece funcionar bien esté o no **synchronized**. Pero hay que considerar aspectos como:

1. ¿Modifica el método el estado de variables “críticas” dentro del objeto? Para descubrir si las variables son o no “críticas” hay que determinar si serán leídas o modificadas por otros hilos del programa. (En este caso, la lectura y modificación son casi siempre llevadas a cabo por métodos **synchronized**, con lo que basta con examinar éstos.) En el caso de **paint( )**, no se hace ninguna modificación.

2. ¿Depende el método del estado de estas variables “críticas”? Si un método **synchronized** modifica una variable que usa tu método, entonces es más que deseable hacer que ese método también sea **synchronized**. Basándonos en esto, podría observarse que **tamanioC** se modifica en métodos **synchronized**, y que por consiguiente, **paint( )** debería ser **synchronized**. Sin embargo, aquí se puede preguntar: ¿qué es lo peor que puede ocurrir si se cambiase **tamanioC** durante un **paint( )**? Cuando se vea que no ocurre nada demasiado malo, se puede decidir dejar **paint( )** como no **synchronized** para evitar la sobrecarga extra intrínseca a llamadas a este tipo de métodos.

3. Una tercera pista es darse cuenta de si la versión base de **paint( )** es **synchronized**, que no lo es. Éste no es un argumento sólido, sólo una pista. En este caso, por ejemplo, se ha mezclado un campo que *se modifica* vía métodos **synchronized** (como **tamanioC**) en la fórmula **paint( )** y podría haber cambiado la situación. Nótese, sin embargo, que el ser **synchronized** no se hereda —es decir, si un método es **synchronized** en la clase base, *no* es automáticamente **synchronized** en la versión superpuesta de la clase derivada.

Se ha modificado el código de prueba de **BeanExplosion2** con respecto al del capítulo anterior para demostrar la habilidad multidifusión de **BeanExplosion2** añadiendo oyentes extra.

Bloqueo

Un hilo puede estar en uno de estos cuatro estados:

1. *Nuevo*: se ha creado el objeto hilo pero todavía no se ha arrancado, por lo que no se puede ejecutar.

2. *Ejecutable*: Significa que el hilo *puede* ponerse en ejecución cuando el mecanismo de reparto de tiempos de UCP tenga ciclos disponibles para el hilo. Por consiguiente, el hilo podría estar o no en ejecución, pero no hay nada para evitar que sea ejecutado si el planificador así lo dispone; no está ni muerto ni bloqueado.

3. *Muerto*: la forma normal de morir de un hilo es que finalice su método **run( )**. También se puede llamar a **stop( )**, pero esto lanza una excepción que es una subclase de **Error** (lo que significa que no hay obligación de poner la llamada en un bloque **try**). Recuérdese que el lanzamiento de una excepción debería ser un evento especial y no parte de la ejecución normal de un programa; por consiguiente, en Java 2 se ha abolido el uso de **stop( )**. También hay un método **destroy( )** (que jamás se implementó) al que nunca habría que llamar si puede evitarse, puesto que es drástico y no libera bloqueos sobre los objetos.

4. *Bloqueado:* podría ejecutarse el hilo, pero hay algo que lo evita. Mientras un hilo esté en estado bloqueado, el planificador simplemente se lo salta y no le cede ningún tipo de UCP. Hasta que el hilo no vuelva al estado ejecutable no hará ninguna operación.

Bloqueándose

El estado bloqueado es el más interesante, y merece la pena examinarlo más en detalle. Un hilo puede bloquearse por cinco motivos:

1. Se ha puesto el hilo a dormir llamando a **sleep(milisegundos)**, en cuyo caso no se ejecutará durante el tiempo especificado.

2. Se ha suspendido la ejecución del hilo con **suspend( )**. No se volverá ejecutable de nuevo hasta que el hilo reciba el mensaje **resume( )**. (Éstos están en desuso en Java 2, y se examinarán más adelante.)

3. Se ha suspendido la ejecución del hilo con **wait( )**. No se volverá ejecutable de nuevo hasta que el hilo reciba los mensajes **notify( )** o **notifyAll( )**. (Sí, esto parece idéntico al caso 2, pero hay una diferencia que luego revelaremos.)

4. El hilo está esperando a que se complete alguna E/S.

5. El hilo está intentando llamar a un método **synchronized** de otro objeto y el bloqueo del objeto no está disponible.

También se puede llamar a **yield( )** (un método de la clase **Thread**) para ceder voluntariamente la UCP de forma que se puedan ejecutar otros hilos. Sin embargo, si el planificador decide que un hilo

ya ha dispuesto de suficiente tiempo ocurre lo mismo, saltándose al siguiente hilo. Es decir, nada evita que el planificador mueva el hilo y le dé tiempo a otro hilo. Cuando se bloquea un hilo, hay alguna razón por la cual no puede continuar ejecutándose.

El ejemplo siguiente muestra las cinco maneras de bloquearse. Todo está en un único archivo denominado **Bloqueo.java**, pero se examinará en fragmentos discretos. (Se verán las etiquetas "Continuará" y "Continuación" que permiten a la herramienta de extracción de código componerlo todo junto.)

Dado que este ejemplo demuestra algunos métodos en desuso, se *obtendrán* mensajes de en "desuso" durante la compilación.

Primero, el marco de trabajo básico:

```
//: c14:Bloqueo.java
// Demuestra las distintas formas en que puede
// bloquearse un hilo.
// <applet code=Bloqueo width=350 height=550>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import com.bruceeckel.swing.*;

//////////// El marco de trabajo básico ///////////
class Bloqueable extends Thread {
  private Elector elector;
  protected JTextField estado = new JTextField(30);
  protected int i;
  public Bloqueable(Container c) {
    c.add(estado);
    elector = new Elector(this, c);
  }
  public synchronized int leer() { return i; }
  protected synchronized void actualizar() {
    estado.setText(getClass().getName()
      + " estado: i = " + i);
  }
  public void pararElector() {
    // elector.stop(); En desuso desde in Java 1.2
    elector.terminar(); // El enfoque preferido
  }
}

class Elector extends Thread {
```

```
  private Bloqueable b;
  private int sesion;
  private JTextField estado = new JTextField(30);
  private boolean parar = false;
  public Elector(Bloqueable b, Container c) {
    c.add(estado);
    this.b = b;
    start();
  }
  public void terminar() { parar = true; }
  public void run() {
    while (!parar) {
      estado.setText(b.getClass().getName()
        + " Elector " + (++sesion)
        + "; valor = " + b.read());
      try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
} ///:Continuará
```

La clase **Bloqueable** pretende ser la clase base de todas las clases en el ejemplo que muestra el bloqueo. Un objeto **Bloqueable** contiene un **JTextField** de nombre **estado** que se usa para mostrar información sobre el objeto. El método que muestra esta información es **actualizar( )**. Se puede ver que usa **getClass( ).getName( )** para producir el nombre de la clase, en vez de simplemente imprimirlo; esto se debe a que **actualizar( )** no puede conocer el nombre exacto de la clase a la que se llama, pues será una clase derivada de **Bloqueable**.

El indicador de cambio de **Bloqueable** es un **int i**, que será incrementado por el método **run( )** de la clase derivada.

Por cada objeto **Bloqueable** se arranca un hilo de clase **Elector**, cuyo trabajo es vigilar a su objeto **Bloqueable** asociado para ver los cambios en **i** llamando a **leer( )** e informando de ellos en su **JTextField estado**. Esto es importante: nótese que **leer( )** y **actualizar( )** son ambos **synchronized**, lo que significa que precisan que el bloqueo del objeto esté libre.

Dormir

La primera prueba del programa se hace con **sleep( )**:

```
///:Continuación
///////////// Bloqueando vía sleep() ///////////
class Durmiente1 extends Bloqueable {
  public Durmiente1(Container c) { super(c); }
```

```
  public synchronized void run() {
    while(true) {
      i++;
      actualizar();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
}

class Durmiente2 extends Bloqueable {
  public Durmiente2(Container c) { super(c); }
  public void run() {
    while(true) {
      cambiar();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
  public synchronized void cambiar() {
      i++;
      actualizar();
  }
} ///:Continuará
```

En **Durmiente1** todo el método **run( )** es **synchronized**. Se verá que el **Elector** asociado con este objeto se ejecutará alegremente *hasta* que el hilo comience, y después el **Elector** se detiene en seco. Ésta es una forma de bloquear: dado que **Durmiente1.run( )** es **synchronized**, y una vez que el objeto empieza, siempre está dentro de **run( )**, el método nunca cede el bloqueo del objeto, quedando **Elector** bloqueado.

Durmiente2 proporciona una solución haciendo **run( )** no **synchronized**. Sólo es **synchronized** el método **cambiar( )**, lo que significa que mientras **run( )** esté en **sleep( )**, el **Elector** puede acceder al método **synchronized** que necesite, en concreto a **leer( )**. Aquí se verá que el **Elector** continúa ejecutándose al empezar el hilo **Durmiente2**.

Suspender y continuar

La siguiente parte del ejemplo presenta el concepto de la suspensión. La clase **Thread** tiene un método **suspend( )** para detener temporalmente el hilo y **resume( )** lo continúa en el punto en el que

se detuvo. Hay que llamar a **resume( )** desde otro hilo fuera del suspendido, y en este caso hay una clase separada denominada **Resumidor** que hace exactamente eso. Cada una de las clases que demuestra suspender/continuar tiene un **Resumidor** asociado:

```
///:Continuación
/////////// Bloqueando vía suspend() ///////////
class SuspenderResumir extends Bloqueable {
  public SuspenderResumir(Container c) {
    super(c);
    new Resumidor(this);
  }
}

class SuspenderResumir1 extends SuspenderResumir {
  public SuspenderResumir1(Container c) { super(c);}
  public synchronized void run() {
    while(true) {
      i++;
      actualizar();
      suspend(); // En desuso desde Java 1.2
    }
  }
}

class SuspenderResumir2 extends SuspenderResumir {
  public SuspenderResumir2(Container c) { super(c);}
  public void run() {
    while(true) {
      cambiar();
      suspend(); // En desuso desde Java 1.2
    }
  }
  public synchronized void cambiar() {
      i++;
      actualizar();
  }
}

class Resumidor extends Thread {
  private SuspenderResumir sr;
  public Resumidor(SuspenderResumir sr) {
    this.sr = sr;
    start();
  }
  public void run() {
```

```
      while(true) {
        try {
          sleep(1000);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
        sr.resume(); // En desuso desde Java 1.2
      }
    }
} ///:Continuará
```

SuspenderResumir1 también tiene un método **syncronized run( )**. De nuevo, al arrancar este hilo se verá que su **Elector** asociado se bloquea esperando a que el bloqueo quede disponible, lo que no ocurre nunca. Esto se fija como antes en **SuspenderResumir2**, en el que no todo el **run( )** es **synchronized**, sino que usa un método **synchronized cambiar( )** diferente.

Hay que ser consciente de que Java 2 ha abolido el uso de **suspend( )** y **resume( )** porque **suspend( )** se guarda el bloqueo sobre el objeto y, por tanto, puede conducir fácilmente a interbloqueos. Es decir, se puede lograr fácilmente que varios objetos bloqueados esperen por culpa de otros que, a su vez, esperan por los primeros, y esto hará que el programa se detenga. Aunque se podrá ver que programas antiguos usan estos métodos, no habría que usarlos. Más adelante, dentro de este capítulo, se describe la solución.

Wait y notify

En los dos primeros ejemplos, es importante entender que, tanto **sleep( )** como **suspend( )**, *no* liberan el bloqueo cuando son invocados. Hay que ser consciente de este hecho si se trabaja con bloqueos. Por otro lado, el método **wait( )** *libera* el bloqueo cuando es invocado, lo que significa que se puede llamar a otros métodos **synchronized** del objeto hilo durante un **wait( )**. En los dos casos siguientes, se verá que el método **run( )** está totalmente **synchronized** en ambos casos, sin embargo, el **Elector** sigue teniendo acceso completo a los métodos **synchronized** durante un **wait( )**. Esto se debe a que **wait( )** libera el bloqueo sobre el objeto al suspender el método en el que es llamado.

También se verá que hay dos formas de **wait( )**. La primera toma como parámetro un número de milisegundos, con el mismo significado que en **sleep( )**: pausar durante ese periodo de tiempo. La diferencia es que en la **wait( )** se libera el bloqueo sobre el objeto *y* se puede salir de **wait( )** gracias a un **notify( )**, o a que una cuenta de reloj expira.

La segunda forma no toma parámetros, y quiere decir que continuará **wait( )** hasta que venga un **notify( )**, no acabando automáticamente tras ningún periodo de tiempo.

Un aspecto bastante único de **wait( )** y **notify( )** es que ambos métodos son parte de la clase base **Object** y no parte de **Thread**, como es el caso de **sleep( )**, **suspend( )** y **resume( )**. Aunque esto parece un poco extraño a primera vista —tener algo que es exclusivamente para los hilos como parte de la clase base universal— es esencial porque manipulan el bloqueo que también es parte de todo objeto. Como resultado, se puede poner un **wait( )** dentro de cualquier método **synchronized**,

independientemente de si hay algún tipo de hilo dentro de esa clase en particular. De hecho, el *único* lugar en el que se puede llamar a **wait( )** es dentro de un método o bloque **synchronized**. Si se llama a **wait( )** o **notify( )** dentro de un método que no es **synchronized**, el programa compilará, pero al ejecutarlo se obtendrá una **IllegalMonitorStateException**, con el mensaje no muy intuitivo de "hilo actual no propietario". Nótese que, desde métodos no **synchronized,** sí que se puede llamar a **sleep( )**, **suspend( )** y **resume( )** puesto que no manipulan el bloqueo.

Se puede llamar a **wait( )** o **notify( )** sólo para nuestro propio bloqueo. De nuevo, se puede compilar código que usa el bloque erróneo, pero producirá el mismo mensaje **IllegalMonitorStateException** que antes. No se puede jugar con el bloqueo de nadie más, pero se puede pedir a otro objeto que lleve a cabo una operación que manipule su propio bloqueo. Por tanto, un enfoque es crear un método **synchronized** que llame a **notify( )** para su propio objeto. Sin embargo, en **Notificador** se verá la llamada **notify( )** dentro de un bloque **synchronized**:

```
synchronized(en2) {
  en2.notify();
}
```

donde **en2** es el tipo de objeto **EsperarNotificar2**. Este método, que no es parte de **EsperarNotificar2**, adquiere el bloqueo sobre el objeto **en2**, instante en el que es legal que invoque al **notify( )** de **en2** sin obtener, por tanto, la **IllegalMonitorStateException**.

```
///Continuación
/////////// Blocking vía wait() ///////////
class EsperarNotificar1 extends Bloqueable {
  public EsperarNotificar1(Container c) { super(c); }
  public synchronized void run() {
    while(true) {
      i++;
      actualizar();
       try {
        wait(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
}

class EsperarNotificar2 extends Bloqueable {
  public EsperarNotificar2(Container c) {
    super(c);
    new Notificador(this);
  }
  public synchronized void run() {
    while(true) {
```

```
        i++;
        actualizar();
         try {
           wait();
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
}

class Notificador extends Thread {
  private EsperarNotificar2 en2;
  public Notificador(EsperarNotificar2 en2) {
    this.en2 = en2;
    start();
  }
  public void run() {
    while(true) {
         try {
          sleep(2000);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
        synchronized(en2) {
          en2.notify();
        }
    }
  }
}  ///: Continuará
```

wait( ) suele usarse cuando se ha llegado a un punto en el que se está esperando alguna otra condición, bajo el control de fuerzas externas al hilo y no se desea esperar ociosamente dentro del hilo. Por tanto, **wait( )** permite poner el hilo a dormir mientras espera a que el mundo cambie, y sólo cuando se da un **notify( )** o un **notifyAll( )** se despierta el método y comprueba posibles cambios. Por consiguiente, proporciona una forma de sincronización entre hilos.

Bloqueo en E/S

Si un flujo está esperando a alguna actividad de E/S, se bloqueará automáticamente. En la siguiente porción del ejemplo, ambas clases funcionan con objetos **Reader** y **Writer**, pero en el marco de trabajo de prueba, se establecerá un flujo entubado para permitir a ambos hilos pasarse datos mutuamente de forma segura (éste es el propósito de los flujos entubados).

El **Emisor** pone datos en el **Writer** y se duerme durante una cantidad de tiempo aleatoria. Sin embargo, **Receptor** no tiene **sleep( )**, **suspend( )** ni **wait( )**. Pero cuando hace un **read( )** se bloquea automáticamente si no hay más datos.

```
///:Continuación
class Emisor extends Bloqueable { // enviar
  private Writer salida;
  public Emisor(Container c, Writer salida) {
    super(c);
    this.salida = salida;
  }
  public void run() {
    while(true) {
      for(char c = 'A'; c <= 'z'; c++) {
        try {
          i++;
          salida.write(c);
          estado.setText("Emisor envio: "
            + (char)c);
          sleep((int)(3000 * Math.random()));
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        } catch(IOException e) {
          System.err.println("Problema de ES");
        }
      }
    }
  }
}
class Receptor extends Bloqueable {
  private Reader entrada;
  public Receptor(Container c, Reader entrada) {
    super(c);
    this.entrada = entrada;
  }
  public void run() {
    try {
      while(true) {
        i++; // Muestra que el elector está vivo
        // Bloquea hasta que los caracteres estén allí:
        estado.setText("Receptor lee: "
          + (char)entrada.read());
      }
    } catch(IOException e) {
      System.err.println("Problema de ES");
```

```
      }
   }
} ///:Continuará
```

Ambas clases también ponen información en sus campos **estado** y cambian **i,** de forma que el **Elector** pueda ver que el hilo se está ejecutando.

Probar

La clase *applet* principal es sorprendentemente simple, porque se ha puesto la mayoría de trabajo en el marco de trabajo **Bloqueable**. Básicamente, se crea un array de objetos **Bloqueable**, y puesto que cada uno es un hilo, llevan a cabo sus propias actividades al presionar el botón "empezar". También hay un botón y una cláusula **actionPerformed( )** para detener todos los objetos **Elector**, que **proporcionan** una demostración de la alternativa al método **stop( )** de Thread, en desuso (en Java 2).

Para establecer una conexión entre los objetos **Emisor** y **Receptor**, se crean un **PipedWriter** y un **PipedReader**. Nótese que el **PipedReader entrada** debe estar conectado al **PipedWriter salida** vía un parámetro del constructor. Después de eso, cualquier cosa que se coloque en **salida** podrá ser después extraída de **entrada**, como si pasara a través de una tubería (y de aquí viene el nombre). Los objetos **entrada** y **salida** se pasan a los constructores **Receptor** y **Emisor** respectivamente, que los tratan como objetos **Reader** y **Writer** de cualquier tipo (es decir, se les aplica un molde hacia arriba).

El array **b** de referencias a **Bloqueable** no se inicializa en este momento de la definición porque no se pueden establecer los flujos entubados antes de que se dé esa definición (lo evita la necesidad del bloque **try**).

```
///:Continuación
/////////// Probando todo ///////////
public class Bloqueo extends JApplet {
   private JButton
      empezar = new JButton("Empezar"),
      pararElectores = new JButton("Detener los Electores");
   private boolean empezado = false;
   private Bloqueable[] b;
   private PipedWriter salida;
   private PipedReader entrada;
   class EmpezarL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         if(!empezado) {
            empezado = true;
            for(int i = 0; i < b.length; i++)
               b[i].start();
         }
      }
   }
   class PararElectoresSL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
```

```
      // Demostración de la alternativa
      // preferida a Thread.stop():
      for(int i = 0; i < b.length; i++)
        b[i].pararElector();
    }
  }
  public void init() {
     Container cp = getContentPane();
     cp.setLayout(new FlowLayout());
     salida = new PipedWriter();
    try {
      entrada = new PipedReader(salida);
    } catch(IOException e) {
      System.err.println("Problema de PipedReader");
    }
    b = new Bloqueable[] {
      new Durmiente1(cp),
      new Durmiente2(cp),
      new SuspenderResumir1(cp),
      new SuspenderResumir2(cp),
      new EsperarNotificar1(cp),
      new EsperarNotificar2(cp),
      new Emisor(cp, salida),
      new Receptor(cp, entrada)
    };
    empezar.addActionListener(new EmpezarL());
    cp.add(empezar);
    pararElectores.addActionListener(
      new PararElectoresL());
    cp.add(pararElectores);
  }
  public static void main(String[] args) {
    Console.run(new Bloqueo(), 350, 550);
  }
} ///:~
```

Nótese en **init( )** el bucle que recorre todo el array añadiendo los campos de texto **estado** y **elector.estado** a la página.

Cuando se crean inicialmente los hilos **Bloqueable**, cada uno crea y arranca automáticamente su propio **Elector**. Por tanto, se verán los **Electores** ejecutándose antes de que se arranquen los hilos **Bloqueable**. Esto es importante, puesto que algunos de los **Electores** se bloquearán y detendrán cuando arranquen los hilos **Bloqueable**, y es esencial ver esto para entender ese aspecto particular del bloqueo.

Interbloqueo

Dado que los hilos pueden bloquearse *y* dado que los objetos pueden tener métodos **synchronized** que evitan que los hilos accedan a ese objeto hasta liberar el bloqueo de sincronización, es posible que un hilo se quede parado esperando a otro, que, de hecho, espera a un tercero, etc. hasta que el último de la cadena resulte ser un hilo que espera por el primero. Se logra un ciclo continuo de hilos que esperan entre sí, de forma que ninguno puede avanzar. A esto se le llama *interbloqueo*. Es verdad que esto no ocurre a menudo, pero cuando le ocurre a uno es muy frustrante.

No hay ningún soporte de lenguaje en Java que ayude a prevenir el interbloqueo; cada uno debe evitarlo a través de un diseño cuidadoso. Estas palabras no serán suficiente para complacer al que esté tratando de depurar un programa con interbloqueos.

La abolición de **stop()**, **suspend( )**, **resume( )** y **destroy()** en Java 2

Uno de los cambios que se ha hecho en Java 2 para reducir la posibilidad de interbloqueo es abolir los métodos **stop( )**, **suspend( )**, **resume( )** y **destroy( )** de **Thread**.

La razón para abolir el método **stop( )** es que no libera los bloqueos que haya adquirido el hilo, y si los objetos están en un estado inconsistente ("dañados") los demás hilos podrán verlos y modificarlos en ese estado. Los problemas resultantes pueden ser grandes y además difíciles de detectar. En vez de usar **stop( )**, habría que seguir el ejemplo de **Bloqueo.java** y usar un indicador (*flag*) que indique al hilo cuando acabar saliendo de su método **run( )**.

Hay veces en que un hilo se bloquea —como cuando se está esperando una entrada— y no puede interrogar al indicador como ocurre en **Bloqueo.java**. En estos casos, se debería seguir sin usar el método **stop( )**, sino usar el método **interrupt( )** de **Thread** para salir del código bloqueado:

```
//: c14:Interrumpir.java
// El enfoque alternativo a usar
// stop() cuando se bloquea un hilo.
// <applet code=Interrumpir width=200 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Bloqueado extends Thread {
  public synchronized void run() {
    try {
      wait(); // Se bloquea
    } catch(InterruptedException e) {
      System.err.println("Interrumpido");
    }
    System.out.println("Saliendo de run()");
```

```
    }
  }

public class Interrumpir extends JApplet {
  private JButton
    interrumpir = new JButton("Interrumpido");
  private Bloqueado bloqueado = new Bloqueado();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(interrumpir);
    interrumpir.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          System.out.println("Boton presionado");
          if(bloqueado == null) return;
          Thread eliminar = bloqueado;
          bloqueado = null; // para liberarlo
          eliminar.interrumpir();
        }
      });
    bloqueado.start();
  }
  public static void main(String[] args) {
    Console.run(new Interrumpir(), 200, 100);
  }
} ///:~
```

El **wait( )** de dentro de **Bloqueado.run( )** produce el hilo bloqueado. Al presionar el botón, se pone a **null** la referencia **bloqueado** de forma que será limpiada por el recolector de basura, y se invoca al método **interrupt( )** del objeto. La primera vez que se presione el botón se verá que el hilo acaba, pero una vez que no hay hilos que matar, simplemente hay que ver que se ha presionado el botón.

Los métodos **suspend( )** y **resume( )** resultan ser inherentemente causantes de interbloqueos. Cuando se llama a **suspend( )**, se detiene el hilo destino, pero sigue manteniendo los bloqueos que haya adquirido hasta ese momento. Por tanto, ningún otro hilo puede acceder a los recursos bloqueados, hasta que el hilo continúe. Cualquier hilo que desee continuar el hilo destino, y que también intente usar cualquiera de los recursos bloqueados, producirá interbloqueo. No se debería usar **suspend( )** y **resume( )**, sino que en su lugar se pone un indicador en la clase **Thread** para indicar si debería activarse o suspenderse el hilo. Si el *flag* indica que el hilo está suspendido, el hilo se mete en una espera usando **wait ( )**. Cuando el *flag* indica que debería continuarse el hilo, se reinicia éste con **notify( )**. Se puede producir un ejemplo modificando **Contador2.java**. Aunque el efecto es similar, se verá que la organización del código es bastante diferente —se usan clases internas anónimas para

todos los oyentes y el **Thread** es una clase interna, lo que hace la programación ligeramente más conveniente, puesto que elimina parte de la contabilidad necesaria en **Contador2.java**:

```
//: c14:Suspender.java
// El enfoque alternativo al uso de suspend()
// y resume(), abolidos en Java 2.
// <applet code=Suspender width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Suspender extends JApplet {
  private JTextField t = new JTextField(10);
  private JButton
    suspender = new JButton("Suspender"),
    resumir = new JButton("Continuar");
  private Suspendible ss = new Suspendible();
  class Suspendible extends Thread {
    private int conteo = 0;
    private boolean suspendido = false;
    public Suspendible() { start(); }
    public void fauxSuspender() {
      suspendido = true;
    }
    public synchronized void fauxResumir() {
      suspendido = false;
      notify();
    }
    public void run() {
      while (true) {
        try {
          sleep(100);
          synchronized(this) {
            while(suspendido)
              wait();
          }
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
        t.setText(Integer.toString(conteo++));
      }
    }
  }
```

```
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        suspender.addActionListener(
            new ActionListener() {
                public
                void actionPerformed(ActionEvent e) {
                    ss.fauxSuspender();
                }
            });
        cp.add(suspender);
        resumir.addActionListener(
            new ActionListener() {
                public
                void actionPerformed(ActionEvent e) {
                    ss.fauxResumir();
                }
            });
        cp.add(resumir);
    }
    public static void main(String[] args) {
        Console.run(new Suspender(), 300, 100);
    }
} ///:~
```

El indicador **suspendido** de **suspendido** se usa para activar y desactivar la suspensión. Para suspender, se pone el indicador a **true** llamando a **fauxSuspender( )**, y esto se detecta dentro de **run( )**. El método **wait( )**, como se describió anteriormente en este capítulo, debe ser **synchronized,** de forma que tenga el bloqueo al objeto. En **fauxResumir( )**, se pone el indicador **suspendedido** a **false** y se llama a **notifiy( )** —puesto que esto despierta a **wait( )** de una cláusula **synchronized**, el método **fauxResumir( )** también debe ser **syncronized( )** de forma que adquiera el bloqueo antes de llamar a **notify( )** (por consiguiente, el bloqueo queda disponible para **wait( )**...). Si se sigue el estilo mostrado en este programa, se puede evitar usar **suspend( )** y **resume( )**.

El método **destroy( )** de **Thread( )** nunca fue implementado; es como un **suspend( )** que no se puede continuar, por lo que tiene los mismos aspectos de interbloqueo que **suspend( )**. Sin embargo, éste no es un método abolido y puede que se implemente en una versión futura de Java (posterior a la 2) para situaciones especiales en las que el riesgo de interbloqueo sea aceptable.

Uno podría preguntarse por qué estos métodos, ahora abolidos, se incluyeron en Java en primer lugar. Parece admitir un error bastante importante para simplemente eliminarlas (e introduciendo otro agujero más en los argumentos que hablan del excepcional diseño de Java y de su infalibilidad, de los que tanto hablaban los encargados de marketing en Sun). Lo más alentador del cambio es que indica claramente que es el personal técnico y no el de marketing el que dirige el asunto —descubrieron el problema y lo están arreglando. Creemos que esto es mucho más prometedor y alen-

tador que dejar el problema ahí pues "corregirlo supondría admitir un error". Esto significa que Java continuará mejorando, incluso aunque esto conlleve alguna pequeña molestia para los programadores de Java. Preferimos, no obstante, afrontar estas molestias a ver cómo se estanca el lenguaje.

Prioridades

La *prioridad* de un hilo indica al planificador lo importante que es cada hilo. Si hay varios hilos bloqueados o en espera de ejecutarse, el planificador ejecutará el de mayor prioridad en primer lugar. Sin embargo, esto no significa que los hilos de menor prioridad no se ejecuten (es decir, no se llega a interbloqueo simplemente con la aplicación directa de estos principios). Los hilos de menor prioridad tienden a ejecutarse menos a menudo.

Aunque es interesante conocer las prioridades que se manejan, en la práctica casi nunca hay que establecer a mano las prioridades. Por tanto uno puede saltarse el resto de esta sección si no le interesan las prioridades.

Leer y establecer prioridades

Se puede leer la prioridad de un hilo con **getPriority( )** y cambiarla con **setPriority( )**. Se puede usar la forma de los ejemplos "contador" anteriores para mostrar el efecto de variar las prioridades. En este *applet* se verá que los contadores se ralentizan, dado que se han disminuido las prioridades de los hilos asociados:

```
//: c14:Contador5.java
// Ajustando las prioridades de los hilos.
// <applet code=Contador5 width=450 height=600>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Teletipo2 extends Thread {
  private JButton
    b = new JButton("Conmutar"),
   incPrioridad = new JButton("arriba"),
   decPrioridad = new JButton("abajo");
  private JTextField
    t = new JTextField(10),
    pr = new JTextField(3); // Mostrar la prioridad
  private int conteo = 0;
  private boolean flagEjecutar = true;
  public Teletipo2(Container c) {
    b.addActionListener(new ConmutadorL());
```

```
      incPrioridad.addActionListener(new ArribaL());
      decPrioridad.addActionListener(new AbajoL());
      JPanel p = new JPanel();
      p.add(t);
      p.add(pr);
      p.add(b);
      p.add(incPrioridad);
      p.add(decPrioridad);
      c.add(p);
    }
    class ConmutadorL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        flagEjecutar = !flagEjecutar;
      }
    }
    class ArribaL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int nuevaPrioridad = getPriority() + 1;
        if(nuevaPrioridad > Thread.MAX_PRIORITY)
          nuevaPrioridad = Thread.MAX_PRIORITY;
        setPriority(nuevaPrioridad);
      }
    }
    class AbajoL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int nuevaPrioridad = getPriority() - 1;
        if(nuevaPrioridad < Thread.MIN_PRIORITY)
          nuevaPrioridad = Thread.MIN_PRIORITY;
        setPriority(nuevaPrioridad);
      }
    }
    public void run() {
      while (true) {
        if(flagEjecutar) {
          t.setText(Integer.toString(conteo++));
          pr.setText(
            Integer.toString(getPriority()));
        }
        yield();
      }
    }
}

public class Contador5 extends JApplet {
  private JButton
```

```
      empezar = new JButton("Empezar"),
      arribaMax = new JButton("Inc Prioridad Max "),
      abajoMax = new JButton("Dec Prioridad Max ");
   private boolean empezado = false;
   private static final int TAMANIO = 10;
   private Teletipo2[] s = new Teletipo2[TAMANIO];
   private JTextField mp = new JTextField(3);
   public void init() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      for(int i = 0; i < s.length; i++)
         s[i] = new Teletipo2(cp);
      cp.add(new JLabel(
         "MAX_PRIORITY = " + Thread.MAX_PRIORITY));
      cp.add(new JLabel("MIN_PRIORITY = "
         + Thread.MIN_PRIORITY));
      cp.add(new JLabel("Agrupar Prioridad Max = "));
      cp.add(mp);
      cp.add(empezar);
      cp.add(arribaMax);
      cp.add(abajoMax);
      empezar.addActionListener(new EmpezarL());
      arribaMax.addActionListener(new ArribaMaxL());
      abajoMax.addActionListener(new AbajoMaxL());
      mostrarMaxPrioridad();
      // Mostrar recursivamente los grupos de hilos padre:
      ThreadGroup padre =
         s[0].getThreadGroup().getParent();
      while(padre != null) {
         cp.add(new Label(
            "Max prioridad del grupo de hilos del padre = "
            + padre.getMaxPriority()));
         padre = padre.getParent();
      }
   }
   public void mostrarMaxPrioridad() {
      mp.setText(Integer.toString(
         s[0].getThreadGroup().getMaxPriority()));
   }
   class EmpezarL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         if(!empezado) {
            empezado = true;
            for(int i = 0; i < s.length; i++)
               s[i].start();
```

```
            }
        }
    }
    class ArribaMaxL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            int maxp =
                s[0].getThreadGroup().getMaxPriority();
            if(++maxp > Thread.MAX_PRIORITY)
                maxp = Thread.MAX_PRIORITY;
            s[0].getThreadGroup().setMaxPriority(maxp);
            mostrarMaxPrioridad();
        }
    }
    class AbajoMaxL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            int maxp =
                s[0].getThreadGroup().getMaxPriority();
            if(--maxp < Thread.MIN_PRIORITY)
                maxp = Thread.MIN_PRIORITY;
            s[0].getThreadGroup().setMaxPriority(maxp);
            mostrarMaxPrioridad();
        }
    }
    public static void main(String[] args) {
        Console.run(new Contador5(), 450, 600);
    }
} ///:~
```

Teletipo2 sigue la forma establecida previamente en este capítulo, pero hay un **JTextField** extra para mostrar la prioridad del hilo y dos botones más para incrementar y disminuir la prioridad.

Hay que tener en cuenta el uso de **yield( )**, que devuelve automáticamente el control al planificador. Sin éste, el mecanismo de multihilos sigue funcionando, pero se verá que funciona lentamente (puede intentarse eliminar la llamada a **yield( )** para comprobarlo). También se podría llamar a **sleep( )**, pero entonces el ratio de cuenta estaría controlado por la duración de **sleep( )** en vez de por la prioridad.

El **init( )** de **Contador5** crea un array de diez **Teletipos2**; sus botones y campos los ubica en el formulario el constructor de **Teletipo2**. **Contador5** añade botones para dar comienzo a todo además de incrementar y disminuir la prioridad máxima del grupo de hilos. Además, están las etiquetas que muestran las prioridades máxima y mínima posibles para un hilo, y un **JTextField** para mostrar la prioridad máxima del grupo de hilos. (La siguiente sección describirá los grupos de hilos). Finalmente, también se muestran como etiquetas las prioridades de los grupos de hilos padre.

Cuando se presiona un botón "arriba" o "abajo", se alcanza esa prioridad de **Teletipo2**, que es incrementada o disminuida de forma acorde.

Cuando se ejecute este programa, se apreciarán varias cosas. En primer lugar, la prioridad por defecto del grupo de hilos es cinco. Incluso si se disminuye la prioridad máxima por debajo de cinco antes de que los hilos empiecen (o antes de crearlos, lo que requiere cambiar el código), cada hilo tendrá su prioridad por defecto a cinco.

La prueba más sencilla es tomar un contador y disminuir su prioridad a uno, y observar que cuenta mucho más lentamente. Pero ahora puede intentarse incrementarla de nuevo. Se puede volver a la prioridad del grupo de hilos, pero no a ninguna superior. Ahora puede disminuirse un par de veces la prioridad de un grupo de hilos. Las prioridades de los hilos no varían, pero si se intenta modificar éstas hacia arriba o hacia abajo, se verá que sacan automáticamente la prioridad del grupo de hilos. Además, se seguirá dando a los hilos nuevos una prioridad por defecto, incluso aunque sea más alta que la prioridad del grupo. (Por consiguiente la prioridad del grupo no es una forma de evitar que los hilos nuevos tengan prioridades mayores a las de los existentes.)

Finalmente, puede intentarse incrementar la prioridad máxima del grupo. No puede hacerse. Las prioridades máximas de los grupos de hilos sólo pueden reducirse, nunca incrementarse.

Grupos de hilos

Todos los hilos pertenecen a un grupo de hilos. Éste puede ser, o bien el grupo de hilos por defecto, o un grupo explícitamente especificado al crear el hilo. En el momento de su creación, un hilo está vinculado a un grupo y no puede cambiar a otro distinto. Cada aplicación tiene, al menos, un hilo que pertenece al grupo de hilos del sistema. Si se crean más hilos sin especificar ningún grupo, éstos también pertenecerán al grupo de hilos del sistema.

Los grupos de hilos también deben pertenecer a otros grupos de hilos. El grupo de hilos al que pertenece uno nuevo debe especificarse en el constructor. Si se crea un grupo de hilos sin especificar el grupo de hilos al que pertenezca, se ubicará bajo el grupo de hilos del sistema. Por consiguiente, todos los grupos de hilos de la aplicación tendrán como último padre al grupo de hilos del sistema.

La razón de la existencia de grupos de hilos no es fácil de determinar a partir de la literatura, que tiende a ser confusa en este aspecto. Suelen citarse "razones de seguridad". De acuerdo con Arnold & Gosling[2], "Los hilos de un grupo de hilos pueden modificar a los otros hilos del grupo, incluyendo todos sus descendientes. Un hilo no puede modificar hilos de fuera de su grupo o grupos contenidos." Es difícil saber qué se supone que quiere decir en este caso el término "modificar". El ejemplo siguiente muestra un hilo en un subgrupo "hoja", modificando las prioridades de todos los hilos en su árbol de grupos de hilos, además de llamar a un método para todos los hilos de su árbol.

```
//: c14:PruebaAcceso.java
// Como los hilos pueden acceder a otros hilos de
// un grupo de hilos padre.
```

[2] *The Java Programming Language*, de Ken Arnold y James Gosling, Addison Wesley 1996 pag. 179.

```
public class PruebaAcceso {
  public static void main(String[] args) {
    ThreadGroup
      x = new ThreadGroup("x"),
      y = new ThreadGroup(x, "y"),
      z = new ThreadGroup(y, "z");
    Thread
      uno = new PruebaHilo1(x, "uno"),
      dos = new PruebaHilo2(z, "dos");
  }
}

class PruebaHilo1 extends Thread {
  private int i;
  PruebaHilo1(ThreadGroup g, String nombre) {
    super(g, nombre);
  }
  void f() {
    i++; // modificar este hilo
    System.out.println(getName() + " f()");
  }
}

class PruebaHilo2 extends PruebaHilo1 {
  PruebaHilo2(ThreadGroup g, String nombre) {
    super(g, nombre);
    start();
  }
  public void run() {
    ThreadGroup g =
      getThreadGroup().getParent().getParent();
    g.list();
    Thread[] gTodos = new Thread[g.activeCount()];
    g.enumerate(gAll);
    for(int i = 0; i < gTodos.length; i++) {
      gTodos[i].setPriority(Thread.MIN_PRIORITY);
      ((PruebaHilo1)gTodos[i]).f();
    }
    g.list();
  }
} ///:~
```

En el método **main( )** se crean varios **ThreadGroups**, colgando unos de otros: **x** no tiene más parámetros que su nombre (un **String**), por lo que se ubica automáticamente en el grupo de hilos "del

sistema", mientras que **y** está bajo **x,** y **z** está bajo **y**. Nótese que la inicialización se da exactamente en el orden textual, por lo que este código es legal.

Se crean dos hilos y se ubican en distintos grupos de hilos. **PruebaHilo1** no tiene un método **run( )** pero tiene un **f( )** que modifica el hilo e imprime algo, de forma que pueda verse que fue invocado. **PruebaHilo2** es una subclase de **PruebaHilo1**, y su **run( )** está bastante elaborado. Primero toma el grupo de hilos del hilo actual, después asciende dos niveles por el árbol de herencia usando **getParent( )**. (Esto se hace así porque ubicamos el objeto **PruebaHilo2** dos niveles más abajo en la jerarquía a propósito.) En este momento, se crea un array de referencias a **Thread**s usando el método **activeCount( )** para preguntar cuántos hilos están en este grupo de hilos y en todos los grupos de hilos hijo. El método **enumerate( )** ubica referencias a todos estos hilos en el array **gTodos**, después simplemente recorremos todo el array invocando al método **f( )** de cada hilo, además de modificar la prioridad. Además, un hilo de un grupo de hilos "hoja" modifica los hilos en los grupos de hilos padre.

El método de depuración **list( )** imprime toda la información sobre un grupo de hilos en la salida estándar y es útil cuando se investiga el comportamiento de los grupos de hilos. He aquí la salida del programa:

```
java.lang.ThreadGroup[name=x,maxpri=10]
    Thread[uno,5,x]
    java.lang.ThreadGroup[name=y,maxpri=10]
        java.lang.ThreadGroup[name=z,maxpri=10]
            Thread[dos,5,x]
uno f()
dos f()
java.lang.ThreadGroup[name=x,maxpri=10]
    Thread[uno,1,x]
    java.lang.ThreadGroup[name=y,maxpri=10]
        java.lang.ThreadGroup[name=z,maxpri=10]
            Thread[dos,1,x]
```

El método **list( )** no sólo imprime el nombre de clase de **ThreadGroup** o **Thread**, sino que también imprime el nombre del grupo de hilos y su máxima prioridad. En el caso de los hilos, se imprime el nombre del hilo, seguido de la prioridad del hilo y el grupo al que pertenece. Nótese que **list( )** va indentando los hilos y grupos de hilos para indicar que son hijos del grupo no indentado.

Se puede ver que el método **run( )** de **PruebaHilo2** llama a **f( )**, por lo que es obvio que todos los hilos de un grupo sean vulnerables. Sin embargo, se puede acceder sólo a los hilos que se ramifican del propio árbol de grupos de hilos **sistema**, y quizás a esto hace referencia el término "seguridad". No se puede acceder a ningún otro árbol de grupos de hilos del sistema.

Controlar los grupos de hilos

Dejando de lado los aspectos de seguridad, algo para lo que los grupos de hilos parecen ser útiles es para controlar: se pueden llevar a cabo ciertas operaciones en todo un grupo de hilos con un único comando. El ejemplo siguiente lo demuestra, además de las restricciones de prioridades en los

grupos de hilos. Los números comentados entre paréntesis proporcionan una referencia para comparar la salida.

```
//: c14:GrupoHilos1.java
// Cómo los grupos de hilos controlan las prioridades
// de los hilos que contienen.

public class GrupoHilos1 {
  public static void main(String[] args) {
    // Lograr el hilo del sistema e imprimir su Info:
    ThreadGroup sis =
      Thread.currentThread().getThreadGroup();
    sis.list(); // (1)
    // Reducir la prioridad del grupo de hilos system:
    sis.setMaxPriority(Thread.MAX_PRIORITY - 1);
    // Incrementar la prioridad del hilo principal:
    Thread curr = Thread.currentThread();
    curr.setPriority(curr.getPriority() + 1);
    sis.list(); // (2)
    // Intentar poner un grupo a la prioridad max:
    ThreadGroup g1 = new ThreadGroup("g1");
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    // Intentar poner un hilo nuevo a prioridad max:
    Thread t = new Thread(g1, "A");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (3)
    // Reducir la prioridad max de g1, después intentar
    // aumentarla:
    g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    g1.list(); // (4)
    // Intentar poner un hilo nuevo a prioridad max:
    t = new Thread(g1, "B");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (5)
    // Bajar la prioridad max por debajo de la prioridad
    // por defecto de los hilos:
    g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
    // Mirar la prioridad de un hilo antes y
    // después de cambiarla:
    t = new Thread(g1, "C");
    g1.list(); // (6)
    t.setPriority(t.getPriority() -1);
    g1.list(); // (7)
    // Hacer que g2 sea un Threadgroup hijo de g1 e
```

```
        // intentar aumentar su prioridad:
        ThreadGroup g2 = new ThreadGroup(g1, "g2");
        g2.list(); // (8)
        g2.setMaxPriority(Thread.MAX_PRIORITY);
        g2.list(); // (9)
        // Añadir un conjunto de hilos nuevos a g2:
        for (int i = 0; i < 5; i++)
          new Thread(g2, Integer.toString(i));
        // Mostrar info sobre todos los grupos de hilos
        // e hilos:
        sis.list(); // (10)
        System.out.println("Comenzando todos los hilos:");
        Thread[] todos = new Thread[sis.activeCount()];
        sis.enumerate(todos);
        for(int i = 0; i < all.length; i++)
          if(!todos[i].isAlive())
            todos[i].start();
        // Suspende & Detiene todos los hilos de
        // este grupo y sus subgrupos:
        System.out.println("Todos los hilos arrancados");
        sis.suspend(); // Abolido en Java 2
        // Aquí nunca se llega...
        System.out.println("Todos los hilos suspendidos");
        sis.stop(); // Abolido en Java 2
        System.out.println("Todos los hilos detenidos");
    }
} ///:~
```

La salida que sigue se ha editado para permitir que entre en una página (se ha eliminado el **java.lang.**), y para añadir números que corresponden a los números en forma de comentario del listado de arriba.

```
(1)  ThreadGroup[name=system,maxpri=10]
         Thread[main,5,system]
(2)  ThreadGroup[name=system,maxpri=9]
         Thread[main,6,system]
(3)  ThreadGroup[name=g1,maxpri=9]
         Thread[A,9,g1]
(4)  ThreadGroup[name=g1,maxpri=8]
         Thread[A,9,g1]
(5)  ThreadGroup[name=g1,maxpri=8]
         Thread[A,9,g1]
         Thread[B,8,g1]
(6)  ThreadGroup[name=g1,maxpri=3]
         Thread[A,9,g1]
         Thread[B,8,g1]
```

```
        Thread[C,6,g1]
(7)   ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,3,g1]
(8)   ThreadGroup[name=g2,maxpri=3]
(9)   ThreadGroup[name=g2,maxpri=3]
(10)  ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
        ThreadGroup[name=g1,maxpri=3]
          Thread[A,9,g1]
          Thread[B,8,g1]
          Thread[C,3,g1]
          ThreadGroup[name=g2,maxpri=3]
            Thread[0,6,g2]
            Thread[1,6,g2]
            Thread[2,6,g2]
            Thread[3,6,g2]
            Thread[4,6,g2]
Comenzando todos los hilos
Todos los hilos arrancados
```

Todos los programas tienen al menos un hilo en ejecución, y lo primero que hace el método **main( )** es llamar al método **static** de **Thread** llamado **currentThread( )**. Desde este hilo, se produce el grupo de hilos y se llama a **list( )** para obtener el resultado. La salida es:

```
(1)   ThreadGroup[name=system,maxpri=10]
        Thread[main,5,system]
```

Puede verse que el nombre del grupo de hilos principal es **system**, y el nombre del hilo principal es **main**, y pertenece al grupo de hilos **system**.

El segundo ejercicio muestra que se puede reducir la prioridad máxima del grupo **system**, y que es posible incrementar la prioridad del hilo **main**:

```
(2)   ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
```

El tercer ejercicio crea un grupo de hilos nuevo, **g1**, que pertenece automáticamente al grupo de hilos **system,** puesto que no se especifica nada más. Se ubica en **g1** un nuevo hilo **A**. Después de intentar establecer la prioridad máxima del grupo y la prioridad de **A** al nivel más alto el resultado es:

```
(3)   ThreadGroup[name=g1,maxpri=9]
        Thread[A,9,g1]
```

Por consiguiente, no es posible cambiar la prioridad máxima del grupo de hilos para que sea superior a la de su grupo de hilos padre.

El cuarto ejercicio reduce la prioridad máxima de **g1** por la mitad y después trata de incrementarla hasta **Thread.MAX_PRIORITY**. El resultado es:

```
(4)   ThreadGroup[name=g1,maxpri=8]
          Thread[A,9,g1]
```

Puede verse que no funcionó el incremento en la prioridad máxima. La prioridad máxima de un grupo de hilos sólo puede disminuirse, no incrementarse. También, nótese que la prioridad del hilo **A** no varió, y ahora es superior a la prioridad máxima del grupo de hilos. Cambiar la prioridad máxima de un grupo de hilos no afecta a los hilos existentes.

El quinto ejercicio intenta establecer como prioridad de un hilo la prioridad máxima:

```
(5)   ThreadGroup[name=g1,maxpri=8]
          Thread[A,9,g1]
          Thread[B,8,g1]
```

No se puede cambiar el nuevo hilo a nada superior a la prioridad máxima del grupo de hilos.

La prioridad por defecto para los hilos de este programa es seis; esa es la prioridad en la que se crearán los hilos nuevos y en la que éstos permanecerán mientras no se manipule su prioridad. El Ejercicio 6 disminuye la prioridad máxima del grupo de hilos por debajo de la prioridad por defecto para ver qué ocurre al crear un nuevo hilo bajo esta condición:

```
(6)   ThreadGroup[name=g1,maxpri=3]
          Thread[A,9,g1]
          Thread[B,8,g1]
          Thread[C,6,g1]
```

Incluso aunque la prioridad máxima del grupo de hilos es tres, el hilo nuevo se sigue creando usando la prioridad por defecto de seis. Por consiguiente, la prioridad máxima del grupo de hilos no afecta a la prioridad por defecto. (De hecho, parece no haber forma de establecer la prioridad por defecto de los hilos nuevos.)

Después de cambiar la prioridad, al intentar disminuirla en una unidad, el resultado es:

```
(7)   ThreadGroup[name=g1,maxpri=3]
          Thread[A,9,g1]
          Thread[B,8,g1]
          Thread[C,3,g1]
```

La prioridad máxima de los grupos de hilos sólo se ve reforzada al intentar cambiar la prioridad.

En (8) y (9) se hace un experimento semejante creando un nuevo grupo de hilos **g2** como hijo de **g1** y cambiando su prioridad máxima. Puede verse que es imposible que la prioridad máxima de **g2** sea superior a la de **g1**:

```
(8)   ThreadGroup[name=g2,maxpri=3]
(9)   ThreadGroup[name=g2,maxpri=3]
```

Nótese que la prioridad de **g2** se pone automáticamente en la prioridad máxima del grupo de hilos **g1** en el momento de su creación.

Después de todos estos experimentos, se imprime la totalidad del sistema de grupos de hilos e hilos:

```
(10) ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
        ThreadGroup[name=g1,maxpri=3]
           Thread[A,9,g1]
           Thread[B,8,g1]
           Thread[C,3,g1]
           ThreadGroup[name=g2,maxpri=3]
              Thread[0,6,g2]
              Thread[1,6,g2]
              Thread[2,6,g2]
              Thread[3,6,g2]
              Thread[4,6,g2]
```

Por tanto, debido a las reglas de los grupos de hilos, un grupo hijo debe tener siempre una prioridad máxima inferior o igual a la prioridad máxima de su padre.

La última parte de este programa demuestra métodos para grupos completos de hilos. En primer lugar, el programa recorre todo el árbol de hilos y pone en marcha todos los que no hayan empezado. Después se suspende y finalmente se detiene el grupo **system**. (Aunque es interesante ver que **suspend( )** y **stop( )** afectan a todo el grupo de hilos, habría que recordar que estos métodos se han abolido en Java 2.) Pero cuando se suspende el grupo **system** también se suspende el hilo **main,** apagando todo el programa, por lo que nunca llega al punto en el que se detienen todos los hilos. De hecho, si no se detiene el hilo **main,** éste lanza una excepción **ThreadDeath**, lo cual no es lo más habitual. Puesto que **ThreadGroup** se hereda de **Object**, que contiene el método **wait( ),** también se puede elegir suspender el programa durante unos segundos invocando a **wait(segundos*1000)**. Por supuesto éste debe adquirir el bloqueo dentro de un bloqueo sincronizado.

La clase **ThreadGroup** también tiene métodos **suspend( )** y **resume( )** por lo que se puede parar y arrancar un grupo de hilos completo y todos sus hilos y subgrupos con un único comando. (De nuevo, **suspend( )** y **resume( )** están en desuso en Java 2.)

Los grupos de hilos pueden parecer algo misteriosos a primera vista, pero hay que tener en cuenta que probablemente no se usarán a menudo directamente.

Volver a visitar Runnable

Anteriormente en este capítulo, sugerimos que se pensara detenidamente antes de hacer un *applet* o un **Frame** principal como una implementación de **Runnable**. Por supuesto, si hay que heredar de una clase *y* se desea añadir comportamiento basado en hilos a la clase, la solución correcta es **Runnable**. El ejemplo final de este capítulo explota esto construyendo una clase **Runnable JPanel** que

pinta distintos colores por sí misma. Esta aplicación toma valores de la línea de comandos para determinar cuán grande es la rejilla de colores y cuán largo es el **sleep( )** entre los cambios de color. Jugando con estos valores se descubrirán algunas facetas interesantes y posiblemente inexplicables de los hilos:

```
//: c14:CajasColores.java
// Usando la interfaz Runnable.
// <applet code=CajasColores width=500 height=400>
// <param name=rejilla value="12">
// <param name=pausa value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class CajaC extends JPanel implements Runnable {
  private Thread t;
  private int pausa;
  private static final Color[] colores = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color colorC = nuevoColor();
  private static final Color nuevoColor() {
    return colores[
      (int)(Math.random() * colores.length)
    ];
  }
  public void paintComponent(Graphics  g) {
    super.paintComponent(g);
    g.setColor(colorC);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  public CajaC(int pausa) {
    this.pausa = pausa;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    while(true) {
```

```
            colorC = nuevoColor();
            repaint();
            try {
              t.sleep(pause);
            } catch(InterruptedException e) {
              System.err.println("Interrumpido");
            }
        }
    }
}

public class CajasColores extends JApplet {
  private boolean esApplet = true;
  private int rejilla = 12;
  private int pausa = 50;
  public void init() {
    // Tomar los parámetros de la página Web:
    if (esApplet) {
      String tamanioR = getParameter("rejilla");
      if(tamanioR != null)
        rejilla = Integer.parseInt(tamanioR);
      String psa = getParameter("pausa");
      if(psa != null)
        pausa = Integer.parseInt(psa);
    }
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(rejilla, rejilla));
    for (int i = 0; i < rejilla * rejilla; i++)
      cp.add(new CajaC(pausa));
  }
  public static void main(String[] args) {
    CajasColores applet = new CajasColores();
    applet.esApplet = false;
    if(args.length > 0)
      applet.rejilla = Integer.parseInt(args[0]);
    if(args.length > 1)
      applet.pausa = Integer.parseInt(args[1]);
    Console.run(applet, 500, 400);
  }
} ///:~
```

CajasColores es la aplicación/*applet* habitual con un **init( )** que establece el IGU. Éste crea **GridLayout,** de forma que tenga celdas **rejilla** en cada dimensión. Después, añade el número apropiado de objetos **CajaC** para rellenar la rejilla, pasando el valor **pausa** a cada uno. En el método **main( )** pue-

de verse que **pausa** y **rejilla** tienen valores por defecto que pueden cambiarse si se pasan parámetros de línea de comandos, o usando parámetros del applet.

Todo el trabajo se da en **CajaC**. Ésta se hereda de **JPanel** e implementa la interfaz **Runnable** de forma que cada **JPanel** también puede ser un **Thread**. Recuérdese que al implementar **Runnable** no se hace un objeto **Thread,** sino simplemente una clase con un método **run( )**. Por consiguiente, un objeto **Thread** hay que crearlo explícitamente y pasarle el objeto **Runnable** al constructor, después llamar a **start( )** (esto ocurre en el constructor). En **CajaC** a este hilo se le denomina **t**.

Fijémonos en el array **colores** que es una enumeración de todos los colores de la clase **Color**. Se usa en **nuevoColor( )** para producir un color seleccionado al azar. El color de la celda actual es **celdaColor**.

El método **paintComponent( )** es bastante simple —simplemente pone el color a **color** y rellena todo el **JPanel** con ese color.

En **run( )** se ve el bucle infinito que establece el **Color** a un nuevo color al azar y después llama a **repaint( )** para mostrarlo. Después el hilo va a **sleep( )** durante la cantidad de tiempo especificada en la línea de comandos.

Precisamente porque este diseño es flexible y la capacidad de hilado está vinculada a cada elemento **JPanel**, se puede experimentar construyendo tantos hilos como se desee. (Realmente, hay una restricción impuesta por la cantidad de hilos que puede manejar cómodamente la JVM.)

Este programa también hace una medición interesante, puesto que puede mostrar diferencias de rendimiento drásticas entre una implementación de hilos de una JVM y otra.

Demasiados hilos

En algún momento, se verá que **CajasColores** se colapsa. En nuestra máquina esto ocurre en cualquier lugar tras una rejilla de 10 + 10. ¿Por qué ocurre esto?

Uno sospecha, naturalmente, que Swing debería estar relacionado con esto, por lo que hay un ejemplo que prueba esa premisa construyendo menos hilos. El siguiente código se ha reorganizado de forma que un **ArrayList implemente Runnable** y un **ArrayList** guarde un número de bloques de colores y elija al azar los que va a actualizar. Después, se crea un número de estos objetos **ArrayList**, dependiendo de la dimensión de la rejilla que se pueda elegir. Como resultado, se tienen bastantes menos hilos que bloques de color, por lo que si se produce un incremento de velocidad se sabrá que se debe a que hay menos hilos que en el ejemplo anterior:

```
//: c14:CajasColores2.java
// Balanceando el uso de hilos.
// <applet code=CajasColores2 width=600 height=500>
// <param name=rejilla value="12">
// <param name=pausa value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

class CajaC2 extends JPanel {
  private static final Color[] colores = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color colorC = nuevoColor();
  private static final Color nuevoColor() {
    return colores[
      (int)(Math.random() * colores.length)
    ];
  }
  void siguienteColor() {
    colorC = nuevoColor();
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(colorC);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
}

class ListaCajaC
  extends ArrayList implements Runnable {
  private Thread t;
  private int pausa;
  public ListaCajaC(int pausa) {
    this.pausa = pausa;
    t = new Thread(this);
  }
  public void comenzar() { t.start(); }
  public void run() {
    while(true) {
      int i = (int)(Math.random() * size());
      ((CajaC2)get(i)).nextColor();
      try {
        t.sleep(pausa);
```

```
            } catch(InterruptedException e) {
               System.err.println("Interrumpido");
            }
         }
      }
   public Object ultimo() { return get(size() - 1);}
}

public class CajasColores2 extends JApplet {
   private boolean esApplet = true;
   private int rejilla = 12;
   // Pausa por defecto más corta que CajasColores:
   private int pausa = 50;
   private ListaCajaC[] v;
   public void init() {
      // Tomar los parámetros de la página Web:
      if (esApplet) {
         String tamanioR = getParameter("rejilla");
         if(tamanioR != null)
            rejilla = Integer.parseInt(tamanioR);
         String psa = getParameter("pausa");
         if(psa != null)
            pausa = Integer.parseInt(psa);
      }
      Container cp = getContentPane();
      cp.setLayout(new GridLayout(rejilla, rejilla));
      v = new ListaCajaC[rejilla];
      for(int i = 0; i < rejilla; i++)
         v[i] = new ListaCajaC(pausa);
      for (int i = 0; i < rejilla * rejilla; i++) {
         v[i % rejilla].add(new CajaC2());
         cp.add((ultimo)v[i % rejilla].ultimo());
      }
      for(int i = 0; i < rejilla; i++)
         v[i].comenzar();
   }
   public static void main(String[] args) {
      CajaColores2 applet = new CajaColores2();
      applet.esApplet = false;
      if(args.length > 0)
         applet.rejilla = Integer.parseInt(args[0]);
      if(args.length > 1)
         applet.pausa = Integer.parseInt(args[1]);
      Console.run(applet, 500, 400);
   }
```

```
} ///:~
```

En **CajaColores2** se crea un array de **ListaCajaC** inicializándose para guardar la **rejilla ListasCajaC**, cada uno de los cuales sabe durante cuánto tiempo dormir. Posteriormente se añade un número igual de objetos **CajaC2** a cada **ListaCajaC**, y se dice a cada lista que **comenzar( )**, lo cual pone en marcha el hilo.

CajaC2 es semejante a **CajaC**: se pinta ListaCajaC a sí misma con un color elegido al azar. Pero esto es *todo* lo que hace un **CajaC2**. Toda la gestión de hilos está ahora en **ListaCajaC**.

ListaCajaC también podría haber heredado **Thread** y haber tenido un objeto miembro de tipo **ArrayList**. Ese diseño tiene la ventaja de que los métodos **add( )** y **get( )** podrían recibir posteriormente un argumento específico y devolver tipos de valores en vez de **Objects** genéricos. (También se podrían cambiar sus nombres para que sean más cortos.) Sin embargo, el diseño usado aquí parecía a primera vista requerir menos código. Además, retiene automáticamente todos los demás comportamientos de un **ArrayList**. Con todas las conversiones y paréntesis necesarios para **get( )**, éste podría no ser el caso a medida que crece el cuerpo del código.

Como antes, al implementar **Runnable** no se logra todo el equipamiento que viene con **Thread**, por lo que hay que crear un nuevo **Thread** y pasárselo explícitamente a su constructor para tener algo en **start( )**, como puede verse en el constructor **ListaCajaC** y en **comenzar( )**. El método **run( )** simplemente elige un número de elementos al azar dentro de la lista y llama al **siguienteColor( )** de ese elemento para que elija un nuevo color seleccionado al azar.

Al ejecutar este programa se ve que, de hecho, se ejecuta más rápido y responde más rápidamente (por ejemplo, cuando es interrumpido, se detiene más rápidamente) y no parece saturarse tanto en tamaños de rejilla grandes. Por consiguiente, se añade un nuevo factor a la ecuación de hilado: hay que vigilar para ver que no se tengan "demasiados hilos" (sea lo que sea lo que esto signifique para cada programa y plataforma en particular —aquí, la ralentización de **CajasColores** parece estar causada por el hecho de que sólo hay un hilo que es responsable de todo el pintado, y que se colapsa por demasiadas peticiones). Si se tienen demasiados hilos hay que intentar usar técnicas como la de arriba para "equilibrar" el número de hilos del programa. Si se aprecian problemas de rendimiento en un programa multihilo, hay ahora varios aspectos que examinar:

1. ¿Hay suficientes llamadas a **sleep( )**, **yield( )** y/o **wait( )**?

2. ¿Son las llamadas a **sleep( )** lo suficientemente rápidas?

3. ¿Se están ejecutando demasiados hilos?

4. ¿Has probado distintas plataformas y JVMs?

Aspectos como éste son la razón por la que a la programación multihilo se le suele considerar un arte.

Resumen

Es vital aprender cuándo hacer uso de capacidades multihilo y cuándo evitarlas. La razón principal de su uso es gestionar un número de tareas que al entremezclarse hagan un uso más eficiente del ordenador (incluyendo la habilidad de distribuir transparentemente las tareas a través de múltiples UCP), o ser más conveniente para el usuario. El ejemplo clásico de balanceo de recursos es usar la UCP durante las esperas de E/S. El ejemplo clásico de conveniencia del usuario es monitorizar un botón de "detención" durante descargas largas.

Las desventajas principales del multihilado son:

1. Ralentización durante la espera por recursos compartidos.

2. Sobrecarga adicional de la UCP necesaria para gestionar los hilos.

3. Complejidad sin recompensa, como la tonta idea de tener un hilo separado para actualizar cada elemento de un array.

4. Patologías que incluyen la inanición, la competición y el interbloqueo.

Una ventaja adicional de los hilos es que sustituyen a las conmutaciones de contexto de ejecución "ligera" (del orden de 100 instrucciones) por conmutaciones de contexto de ejecución "pesada" (del orden de miles de instrucciones). Puesto que todos los hilos de un determinado proceso comparten el mismo espacio de memoria, una conmutación de proceso ligera sólo cambia la ejecución del programa y las variables locales. Por otro lado, un cambio de proceso —la conmutación de contexto pesada— debe intercambiar todo el espacio de memoria.

El multihilado es como irrumpir paso a paso en un mundo completamente nuevo y aprender un nuevo lenguaje de programación o al menos un conjunto de conceptos de lenguaje nuevos. Con la apariencia de soporte a hilos, en la mayoría de sistemas operativos de microcomputador han ido apareciendo extensiones para hilos en lenguajes de programación y bibliotecas. En todos los casos, la programación de hilos (1) parece misteriosa y requiere un cambio en la forma de pensar al programar; y (2) parece similar al soporte de hilos en otros lenguajes, por lo que al entender los hilos se entiende una lengua común. Y aunque el soporte de hilos puede hacer que Java parezca un lenguaje más complicado, no hay que echar la culpa a Java. Los hilos son un truco.

Una de las mayores dificultades con los hilos se debe a que, dado que un recurso —como la memoria de un objeto— podría estar siendo compartido por más de un hilo, hay que asegurarse de que múltiples hilos no intenten leer y cambiar ese recurso simultáneamente. Esto requiere de un uso juicioso de la palabra clave **synchronized**, que es una herramienta útil pero que debe ser totalmente comprendida puesto que puede presentar silenciosamente situaciones de interbloqueos.

Además, hay determinado arte en la aplicación de los hilos. Java está diseñado para permitir la creación de tantos objetos como se necesite para solucionar un problema —al menos en teoría. (Crear millones de objetos para un análisis de elementos finitos de ingeniería, por ejemplo, podría no ser práctico en Java.) Sin embargo, parece que hay un límite superior al número de hilos a crear, puesto que en algún momento, un número de hilos más elevado da muestras de colapso. Este pun-

to crítico no se alcanza con varios miles, como en el caso de los objetos, sino en unos pocos cientos, o incluso a veces menos de 1.200. Como a menudo sólo se crea un puñado de hilos para solucionar un problema, este límite no suele ser tal, aunque puede parecer una limitación en diseños generales.

Un aspecto significativo y no intuitivo de los hilos es que, debido a la planificación de los hilos, se puede hacer que una aplicación se ejecute generalmente *más rápido* insertando llamadas a **sleep( )** dentro del bucle principal de **run( )**. Esto hace, sin duda, que su uso parezca un arte, especialmente cuando los retrasos más largos parecen incrementar el rendimiento. Por supuesto, la razón por la que ocurre esto es que retrasos más breves pueden hacer que la interrupción del planificador del final del **sleep( )** se dé antes de que el hilo en ejecución esté listo para ir a dormir, forzando al planificador a detenerlo y volver a arrancarlo más tarde para que pueda acabar con lo que estaba haciendo, para ir después a dormir. Hay que pensar bastante para darse cuenta en lo complicadas que pueden ser las cosas.

Algo que alguien podría echar de menos en este capítulo es un ejemplo de animación, que es una de las cosas más populares que se hacen con los *applets*. Sin embargo, con el Java JDK (disponible en *http://java.sun.com*) viene la solución completa (con sonido) a este problema, dentro de la sección de demostración. Además, se puede esperar que en las versiones futuras de Java se incluya un mejor soporte para animaciones, a la vez que están apareciendo distintas soluciones de animación para la Web, no Java, y que no son de programación, que pueden ser superiores a los enfoques tradicionales. Si se desean explicaciones de cómo funcionan las animaciones en Java, puede verse *Core Java 2*, de Horstmann & Cornell, Prentice-Hall, 1997. Para acceder a discusiones más avanzadas en el multihilado, puede verse *Concurrent Programming in Java*, de Doug Lea, Addison-Wesley, 1997, o *Java Threads* de Oaks & Wong, O'Reilly, 1997.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Heredar una clase de **Thread** y superponer el método **run( )**. Dentro de **run( )**, imprimir un mensaje y llamar después a **sleep( )**. Repetir esto tres veces, después de volver de **run( )** finalice. Poner un mensaje de empiece en el constructor y superponer **finalize( )** para imprimir un mensaje de apagado. Hacer una clase hilo separada que llame a **System.gc( )** y **System.runFinalization( )** dentro de **run( )**, imprimiendo un mensaje a medida que lo hace. Hacer varios objetos hilo de ambos tipos y ejecutarlos para ver qué ocurre.

2. Modificar **Compartiendo2.java** para añadir un bloque **synchronized** dentro del método **run ( )** de **DosContadores** en vez de sincronizar todo el método **run( )**.

3. Crear dos subclases **Thread**, una con un **run( )** que empiece, capture la referencia al segundo objeto **Thread** y llame después a **wait( )**. El método **run( )** de la otra clase debería llamar a **notifyAll( )** para el primer hilo, tras haber pasado cierto número de segundos, de forma que el primer hilo pueda imprimir un mensaje.

4. En **Contador5.java** dentro de **Teletipo2**, retirar el **yield( )** y explicar los resultados. Reemplazar el **yield( )** con un **sleep( )** y explicar los resultados.

5. En **grupoHilos.java**, reemplazar la llamada a **sis.suspend( )** con una llamada a **wait( )** para el grupo de hilos, haciendo que espere durante dos segundos. Para que esto funcione correctamente hay que adquirir el bloqueo de **sis** dentro de un bloque **synchronized**.

6. Cambiar **Demonios.java,** de forma que **main( )** tenga un **sleep( )** en vez de un **readLine ( )**. Experimentar con distintos tiempos en la **sleep( )** para ver qué ocurre.

7. En el Capítulo 8, localizar el ejemplo **ControlesInvernadero.java**, que consiste en tres archivos. En **Evento.java**, la clase **Evento** se basa en vigilar el tiempo. Cambiar **Evento** de forma que sea un **Thread**, y cambiar el resto del diseño de forma que funcione con este nuevo **Evento** basado en **Thread**.

8. Modificar el Ejercicio 7, de forma que se use la clase **java.util.Timer** del JDK 1.3 para ejecutar el sistema.

9. A partir de **OndaSeno.java** del Capítulo 13, crear un programa (un applet/aplicación usando la clase **Console**) que dibuje una onda seno animada que parezca desplazarse por la ventana como si fuera un osciloscopio, dirigiendo la animación con un **Thread**. La velocidad de la animación debería controlarse con un control **java.swing.JSlider**.

10. Modificar el Ejercicio 9, de forma que se creen varios paneles onda seno dentro de la aplicación. El número de paneles debería controlarse con etiquetas HTML o parámetros de línea de comando.

11. Modificar el Ejercicio 9, de forma que se use la clase **java.swing.Timer** para dirigir la animación. Nótese la diferencia entre éste y **java.util.Timer**.

15: Computación distribuida

Históricamente, la programación a través de múltiples máquinas ha sido fuente de error, difícil y compleja.

El programador tenía que conocer muchos detalles sobre la red y, en ocasiones, incluso el hardware. Generalmente era necesario comprender las distintas "capas" del protocolo de red, y había muchas funciones diferentes en cada una de las bibliotecas de la red involucradas con el establecimiento de conexiones, el empaquetado y desempaquetado de bloques de información; el reenvío y recepción de esos bloques; y el establecimiento de acuerdos. Era una tarea tremenda.

Sin embargo, la idea básica de la computación distribuida no es tan complicada, y está abstraída de forma muy elegante en las bibliotecas de Java. Se desea:

- Conseguir algo de información de una máquina lejana y moverla a la máquina local, o viceversa. Esto se logra con programación básica de red.
- Conectarse a una base de datos, que puede residir en otra máquina. Esto se logra con la *Java DataBase Connectivity* (JDBC), que es una abstracción alejada de los detalles difíciles y específicos de cada plataforma de SQL (el lenguaje de consulta estructurado o *Structured Query Language* que se usa en la mayoría de transacciones de bases de datos).
- Proporcionar servicios vía un servidor web. Esto se logra con los *servlets* de Java y las *Java Server Pages* (JSP).
- Ejecutar métodos sobre objetos Java que residan en máquinas remotas, de forma transparente, como si estos objetos residieran en máquinas locales. Esto se logra con el *Remote Method Invocation* (RMI) de Java.
- Utilizar código escrito en otros lenguajes, que está en ejecución en otras arquitecturas. Esto se logra usando la *Common Object Request Broker Architecture* (CORBA), soportado directamente por Java.
- Aislar la lógica de negocio de aspectos de conectividad, especialmente en conexiones con bases de datos, incluyendo la gestión de transacciones y la seguridad. Esto se logra usando *Enterprise JavaBeans* (EJB). Los EJB no son una arquitectura distribuida, sino que las aplicaciones resultantes suelen usarse en un sistema cliente-servidor en red.
- Fácilmente, dinámicamente, añadir y quitar dispositivos de una red que representa un sistema local. Esto se logra con Jini de Java.

En este capítulo se dará a cada tema una ligera introducción. Nótese, por favor, que cada tema es bastante voluminoso y de por sí puede ser fuente de libros enteros, por lo que este capítulo sólo pretende familiarizar al lector con estos temas, y no convertirlo en un experto (sin embargo, se puede recorrer un largísimo camino en la programación en red, *servlets* y JSP con la información que se presenta aquí).

Programación en red

Una de las mayores fortalezas de Java es su funcionamiento inocuo en red. Los diseñadores de bibliotecas de red de Java han hecho que trabajar en red sea bastante similar a la escritura y lectura de archivo, excepto en que el "archivo" exista en una máquina remota y la máquina remota pueda decidir exactamente qué desea hacer con la información que se le envía o solicita. Siempre que sea posible, se han abstraído los detalles de la red subyacente, dejándose para la JVM y la instalación de Java que haya en la máquina local. El modelo de programación que se usa es el de un archivo; de hecho, se envuelve la conexión de red (un "socket") con objetos flujo, por lo que se acaban usando las mismas llamadas a métodos que con el resto de flujos. Además, el multihilo inherente a Java es extremadamente útil al tratar con otro aspecto de red: la manipulación simultánea de múltiples conexiones.

Esta sección introduce el soporte de red de Java usando ejemplos fáciles de entender.

Identificar una máquina

Por supuesto, para comunicarse con una máquina desde otra y para asegurarse de estar conectado con una máquina particular, debe haber alguna forma de identificar de forma unívoca las máquinas de una red. Las primeras redes se diseñaron de forma que proporcionaran nombres únicos de máquinas dentro de la red local. Sin embargo, Java trabaja dentro de Internet, lo que requiere una forma de identificar una máquina unívocamente de entre todas las demás máquinas *del mundo*. Esto se logra con la dirección IP (*Internet Protocol*) que puede existir de dos formas:

1. La forma familiar DNS (*Domain Name System*). Nuestro nombre de dominio es **bruceeckel.com,** y de tener un computador denominado **Opus** en nuestro dominio, su nombre de dominio habría sido **Opus.bruceeckel.com**. Éste es exactamente el tipo de nombre que se usa al enviar un correo a la gente, y suele incorporarse en una dirección World Wide Web.

2. Alternativamente, puede usarse la forma del "cuarteto punteado", que consta de cuatro números separados por puntos, como **123.255.28.120**.

En ambos casos, la dirección IP se representa internamente como un número de 32[1] bits (de forma que cada uno de los cuatro números no puede exceder de 255), y se puede lograr un objeto Java especial para que represente este número de ninguna de las formas de arriba, usando el método **static InetAddress.getByName( )** que está en **java.net**. El resultado es un objeto de tipo **InetAddress**, que puede usarse para construir un "socket" como se verá después.

Como un simple ejemplo del uso de **InetAddress.getByName( )**, considérese lo que ocurre si se tiene un proveedor de servicios de Internet (ISP o *Internet Service Provider*) vía telefónica. Cada vez que uno llama, le asignan una dirección IP temporal. Pero mientras se está conectado, la dirección IP de cada uno tiene la misma validez que cualquier otra dirección IP de la red. Si alguien se co-

[1] Esto implica un máximo de algo más de cuatro millones de números, que se está agotando rápidamente. El nuevo estándar para direcciones IP usará un número de 128 bits, que debería producir direcciones IP únicas suficientes para el futuro predecible.

necta a la máquina utilizando su dirección IP puede conectarse a un servidor web o a un servidor FTP en ejecución en tu máquina. Por supuesto, necesitan saber la dirección IP y cómo éste varía cada vez que uno se conecta, ¿cómo puede saberse?

El programa siguiente usa **InetAddress.getByName( )** para producir la dirección IP actual. Para usarla, hay que saber el nombre del computador que se esté usando. En Windows 95/98 hay que ir a "Configuración", "Panel de Control", "Red" y después seleccionar la solapa "Identificación". El campo "Nombre de PC" es el que hay que poner en la línea de comandos:

```
//: c15:QuienSoyYo.java
// Averigua la dirección de red cuando
// se está conectado a Internet.
import java.net.*;

public class QuienSoyYo {
  public static void main(String[] args)
      throws Exception {
    if(args.length != 1) {
      System.err.println(
        "Uso: QuienSoyYo NombrePC");
      System.exit(1);
    }
    InetAddress a =
      InetAddress.getByName(args[0]);
    System.out.println(a);
  }
} ///:~
```

En este caso, la máquina se denomina "pepe". Por tanto, una vez que nos hemos conectado a nuestro ISP ejecutamos el programa:

```
java QuienSoyyo pepe
```

Recibimos un mensaje como éste (por supuesto, la dirección es distinta cada vez):

```
pepe/199.190.87.75
```

Si le decimos a un amigo esta dirección y tenemos un servidor web en ejecución en nuestro PC, éste puede conectarse a la misma acudiendo a la URL *http://199.190.87.75* (sólo mientras continuemos conectados durante esa sesión). Ésta puede ser una forma útil de distribuir información a alguien más, o probar la configuración de un sitio web antes de enviarlo a un servidor "real".

Servidores y clientes

Todo el sentido de una red es permitir a dos máquinas conectarse y comunicarse. Una vez que ambas máquinas se han encontrado mutuamente, pueden tener una conversación bidireccional. Pero,

¿cómo se identifican mutuamente? Es como perderse en un parque de atracciones: una máquina tiene que estar en un lugar y escuchar mientras la otra máquina dice: "Eh, ¿dónde estás?"

A la máquina que "sigue en un sitio" se le denomina el *servidor*, y a la que la busca se le denomina el *cliente*. Esta distinción es importante sólo mientras el cliente está intentando conectarse al servidor. Una vez que se han conectado, se convierte en un proceso de comunicación bidireccional y no vuelve a ser importante el hecho de si alguno desempeñó el papel de servidor y el otro el de cliente.

Por tanto, el trabajo del servidor es esperar una conexión, y ésta se lleva a cabo por parte del objeto servidor especial que se crea. El trabajo del cliente es intentar hacer una conexión al servidor, y esto lo logra el objeto cliente especial que se crea. Una vez que se hace la conexión se verá que en ambos extremos, servidor y cliente, la conexión se convierte mágicamente en un objeto flujo E/S, y a partir de ese momento se puede tratar la conexión como si simplemente se estuviera leyendo y escribiendo a un archivo. Por consiguiente, tras hacer la conexión, simplemente se usarán los comandos de E/S familiares del Capítulo 11. Ésta es una de las mejores facetas del funcionamiento en red de Java.

Probar programas sin una red

Por muchas razones, se podría no tener una máquina cliente, una máquina servidora y una red disponibles para probar los programas. Se podrían estar llevando a cabo ejercicios en una situación parecida a la de una clase, o se podrían estar escribiendo programas que no son aún lo suficientemente estables como para ponerlos en la red. Los creadores del Internet Protocol eran conscientes de este aspecto, y crearon una dirección especial denominada **localhost** para ser una dirección IP "bucle local" para hacer pruebas sin red. La forma genérica de producir esta dirección en Java es:

```
InetAddress addr = InetAddress.getByName(null);
```

Si se pasa un **null** a **getByName( )**, éste pasa por defecto a usar el **localhost**. La **InetAddress** es lo que se usa para referirse a la máquina particular, y hay que producir este nombre antes de seguir avanzando. No se pueden manipular los contenidos de una **InetAddress** (pero se pueden imprimir, como se verá en el ejemplo siguiente). La única forma de crear un **InetAddress** es a través de uno de los siguientes métodos miembro **static** sobrecargados: **getByName( )** (que es el que se usará generalmente), **getAllByName( )** o **getLocalHost( )**.

También se puede producir la dirección del bucle local pasándole el String **localhost**:

```
InetAddress.getByName("localhost").
```

(asumiendo que "localhost" está configurado en la tabla de "hosts" de la máquina), o usando su forma de cuarteto punteado para nombrar el número IP reservado del bucle:

```
InetAddress.getByName("127.0.0.1");
```

Las tres formas producen el mismo resultado.

Puerto: un lugar único dentro de la máquina

Una dirección IP no es suficiente para identificar un único servidor, puesto que pueden existir muchos servidores en una misma máquina. Cada máquina IP también contiene *puertos*, y cuando se establece un cliente o un servidor hay que elegir un puerto en el que tanto el cliente como el servidor acuerden conectarse; si se encuentra alguno, la dirección IP es el barrio, y el puerto es el bar.

El puerto no es una ubicación física en una máquina, sino una abstracción software (fundamentalmente para propósitos de reservas). El programa cliente sabe como conectarse a la máquina vía su dirección IP, pero, ¿cómo se conecta a un servicio deseado (potencialmente uno de muchos en esa máquina)? Es ahí donde aparecen los puertos como un segundo nivel de direccionamiento. La idea es que si se pregunta por un puerto en particular, se está pidiendo el servicio asociado a ese número. La hora del día es un ejemplo simple de un servicio. Depende del cliente el que éste conozca el número de puerto sobre el que se está ejecutando el servicio deseado.

Los servicios del sistema se reservan el uso de los puertos del 1 al 1.024, por lo que no se deberían usar estos números o cualquier otro puerto que se sepa que está en uso. La primera elección para los ejemplos de este libro será el puerto 8080 (en memoria del venerable chip Intel 8080 de 8 bits de nuestro primer computador, una máquina CP/M).

Sockets

El *socket* es la abstracción software que se usa para representar los "terminales" de una conexión entre dos máquinas. Para una conexión dada, hay un socket en cada máquina, y se puede imaginar un "cable" hipotético entre las dos máquinas, estando cada uno de los extremos del "cable" enchufados al socket. Por supuesto, se desconoce completamente el hardware físico y el cableado entre máquinas. Lo fundamental de esta abstracción es que no hay que saber nada más que lo necesario.

En Java, se crea un socket para hacer una conexión a la otra máquina, después se logra un **InputStream** y un **OutputStream** (o, con los convertidores apropiados, un **Reader** y un **Writer**) a partir del socket para poder tratar la conexión como un objeto flujo de E/S. Hay dos clases de socket basadas en flujos: un **ServerSocket** que usa un servidor para "escuchar" eventos entrantes y un **Socket** que usa un cliente para iniciar una conexión. Una vez que un cliente hace una conexión socket, el **ServerSocket** devuelve (vía el método **accept( )**) un **Socket** correspondiente a través del cual tendrán lugar las comunicaciones en el lado servidor. A partir de ese momento se tiene una auténtica conexión **Socket** a **Socket** y se tratan ambos extremos de la misma forma, puesto que *son* iguales. En este momento se usan los métodos **getInputStream( )** y **getOutputStream( )** para producir los objetos **InputStream** y **OutputStream** correspondientes de cada **Socket**. Éstos deben envolverse en espacios de almacenamiento intermedio y clases formateadoras exactamente igual que cualquier otro objeto flujo descrito en el Capítulo 11.

El uso del término **ServerSocket** habría parecido ser otro ejemplo de un esquema confuso de nombres en las bibliotecas de Java. Podría pensarse que habría sido mejor denominar al **ServerSocket**, "ServerConnector" o algo sin la palabra "Socket". También se podría pensar que, tanto **ServerSocket** como **Socket,** deberían ser heredados de alguna clase base común. Sin duda, ambas clases tienen varios métodos en común, pero no son suficientes como para darles a ambos una clase base co-

mún. En vez de ello, el trabajo de **ServerSocket** es esperar hasta que se le conecte otra máquina, y devolver después un **Socket**. Por esto, **ServerSocket** parece no tener un nombre muy adecuado para lo que hace, pues su trabajo no es realmente ser un socket, sino construir un objeto **Socket** cuando alguien se conecta al mismo.

Sin embargo, el **ServerSocket** crea un "servidor" físico o socket oyente en la máquina host. Este socket queda esperando a posibles conexiones entrantes y devuelve un socket de "establecimiento" (con los puntos de finalización local y remoto definidos) vía el método **accept( )**. La parte confusa es que estos dos sockets (el oyente y el establecimiento están asociados con el mismo socket servidor. El socket oyente sólo puede aceptar nuevas peticiones de conexión, y no paquetes de datos. Por tanto, mientras **ServerSocket** no tenga mucho sentido desde el punto de vista programático, sí lo tiene "físicamente".

Cuando se crea un **ServerSocket**, sólo se le da un número de puerto. No hay que dar una dirección IP, porque ya está en la máquina que representa. Sin embargo, cuando se crea un **Socket** hay que dar, tanto una dirección IP como un número de puerto al que se está intentando conectar. (Sin embargo, el **Socket** que vuelve de **ServerSocket.accept( )** ya contiene toda esta información.)

Un servidor y cliente simples

Este ejemplo hace el uso más simple de servidores y clientes basados en sockets. Todo lo que hace el servidor es esperar a una conexión, después usa el **Socket** producido por esa conexión para crear un **InputStream** y un **OutputStream**. Éstos se convierten en un **Reader** y un **Writer**, y después envueltos en un **BufferedReader** y un **PrintWriter**. Después de esto, todo lo que se lee de **Buffered Reader** se reproduce en el **PrintWriter** hasta que recibe la línea "FIN", momento en el que se cierra la conexión.

El cliente hace la conexión al servidor, después crea un **OutputStream** y lleva a cabo la misma envoltura que el servidor. Las líneas de texto se envían a través del **PrintWriter** resultante. El cliente también crea un **InputStream** (de nuevo, con conversiones y envolturas apropiadas) para escuchar lo que esté diciendo el servidor (que, en este caso, es simplemente una reproducción de las mismas palabras).

Tanto el servidor como el cliente usan el mismo número de puerto, y el cliente usa la dirección del bucle local para conectar al servidor en la misma máquina, con lo que no hay que probarlo a través de la red. (En algunas configuraciones, podría ser necesario estar *conectado* a una red para que este programa funcione, incluso si no se está *comunicando* a través de una red.)

He aquí el servidor:

```
//: c15:ServidorParlante.java
// Servidor muy simple que simplemente
// reproduce lo que envía el cliente.
import java.io.*;
import java.net.*;

public class ServidorParlante {
```

```
    // Elegir un Puerto del rango 1-1024:
    public static final int PUERTO = 8080;
    public static void main(String[] args)
        throws IOException {
      ServerSocket s = new ServerSocket(PUERTO);
      System.out.println("Empezado: " + s);
      try {
        // Se bloquea hasta que se dé alguna conexión:
        Socket socket = s.accept();
        try {
          System.out.println(
            "Conexion aceptada: "+ socket);
          BufferedReader entrada =
            new BufferedReader(
              new InputStreamReader(
                socket.getInputStream()));
          // La salida suele hacerse vaciando a ráfagas
          // por parte de PrintWriter:
          PrintWriter salida =
            new PrintWriter(
              new BufferedWriter(
                new OutputStreamWriter(
                  socket.getOutputStream())),true);
          while (true) {
            String str = entrada.readLine();
            if (str.equals("FIN")) break;
            System.out.println("Reproduciendo: " + str);
            salida.println(str);
          }
        // Elegir siempre los dos sockets...
        } finally {
          System.out.println("cerrando...");
          socket.close();
        }
      } finally {
        s.close();
      }
    }
} ///:~
```

Se puede ver que el **ServerSocket** simplemente necesita un número de puerto, no una dirección IP (¡puesto que se está ejecutando en *esta* máquina!). Cuando se llama a **accept( )** el método *se bloquea* hasta que algún cliente intente conectarse al mismo. Es decir, está ahí esperando a una conexión, pero los otros procesos pueden ejecutarse (véase Capítulo 14). Cuando se hace una conexión, **accept( )** devuelve un objeto **Socket** que representa esa conexión.

La responsabilidad de limpiar los sockets se enfoca aquí cuidadosamente. Si falla el constructor **ServerSocket**, el programa simplemente finaliza (nótese que tenemos que asumir que el constructor de **ServerSocket** no deje ningún socket de red abierto sin control si falla). En este caso, **main( ) lanza IOException,** por lo que no es necesario un bloque **try**. Si el constructor **ServerSocket** tiene éxito, hay que guardar otras llamadas a métodos en un bloque **try-finally** para asegurar que, independientemente de cómo se deje el bloque, se cierre correctamente el **ServerSocket**.

Para el **Socket** devuelto por **accept( )** se usa la misma lógica. Si falla **accept( )**, tenemos que asumir que el **Socket** no existe o guarda recursos, por lo que no es necesario limpiarlo. Sin embargo, si tiene éxito, las siguientes sentencias tendrán que estar en un bloque **try-finally,** de forma que si fallan, se seguirá limpiando el **Socket**. Es necesario tener cuidado aquí porque los sockets usan recursos importantes no relacionados con la memoria, por lo que hay que ser diligente para limpiarlos (puesto que no hay ningún destructor que lo haga en Java).

Tanto el **ServerSocket** como el **Socket** producidos por **accept( )** se imprimen a **System.out**. Esto significa que se invoca automáticamente a sus métodos **toString( )**. Éstos producen:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

La siguiente parte del programa parece simplemente como si sólo se abrieran y escribieran archivos, de no ser porque el **InputStream** y **OutputStream** se crean a partir del objeto **Socket**. Tanto el objeto **InputStream** como el objeto **OutputStream** se convierten a objetos **Reader** y **Writer** usando las clases "conversoras" **InputStreamReader** y **OutputStreamWriter**, respectivamente. También se podrían haber usado directamente las clases **InputStream** y **OutputStream** de Java 1.0, pero con la salida, hay una ventaja añadida al usar el enfoque **Writer**. Éste aparece con **PrintWriter**, que tiene un constructor sobrecargado que toma un segundo argumento, un indicador **boolean** que indica si hay que vaciar automáticamente la salida al final de cada sentencia **println( )** (pero *no* **print( )**). Cada vez que se escribe a **out**, hay que vaciar su espacio de almacenamiento intermedio, de forma que la información fluya por la red. El vaciado es importante en este ejemplo particular, porque tanto el cliente como el servidor, esperan ambos a una línea proveniente de la otra parte antes de proceder. Si no se da el vaciado, la información no saldrá a la red hasta llenar el espacio de almacenamiento intermedio, causando muchos problemas en este ejemplo.

Cuando se escriban programas de red hay que tener cuidado con el uso del vaciado automático. Cada vez que se vacía el espacio de almacenamiento intermedio, hay que crear y enviar un paquete. En este caso, eso es exactamente lo que queremos, pues si no se envía el paquete que contiene la línea, se detendría el intercambio amistoso bidireccional entre el cliente y el servidor. Dicho de otra forma, el final de línea es el fin de cada mensaje. Pero en muchos casos, los mensajes no están delimitados por líneas, por lo que es mucho más eficiente no usar el autovaciado y dejar en su lugar que el espacio de almacenamiento intermedio incluido decida cuándo construir y enviar un paquete. De esta forma se pueden enviar paquetes mayores, y el procesamiento será más rápido.

Nótese que, como ocurre con casi todos los flujos que se abren, éstos tienen sus espacios de almacenamiento intermedio. Hay un ejercicio al final de este capítulo para mostrar al lector lo que ocurre si no se utilizan los espacios de almacenamiento intermedio (todo se ralentiza).

El bucle **while** infinito lee líneas del **BufferedReader entrada** y escribe información a **System.out** y al **PrintWriter salida**. Nótese que **entrada** y **salida** podrían ser cualquier flujo, lo que ocurre simplemente es que están conectados a la red.

Cuando el cliente envía la línea "FIN", el programa sale del bucle y cierra el **Socket**.

He aquí el cliente:

```
//: c15:ClienteParlante.java
// Cliente muy simple que sólo envía
// líneas al servidor y lee líneas
// que el servidor envía.
import java.net.*;
import java.io.*;

public class ClienteParlante {
  public static void main(String[] args)
      throws IOException {
    // Pasar null a getByName() genera la dirección
    // IP especial "Loopback Local", que permite
    // hacer pruebas en una máquina con o sin red:
    InetAddress addr =
      InetAddress.getByName(null);
    // De forma alternativa, puedes usar
    // la dirección o nombre:
    // InetAddress addr =
    //    InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    //    InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket =
      new Socket(addr, ServidorParlante.PUERTO);
    // Guardar todo en un try-finally para asegurarse
    // de que se cierra el socket:
    try {
      System.out.println("socket = " + socket);
      BufferedReader entrada =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // La salida es vaciada automáticamente
      // por PrintWriter:
      PrintWriter salida =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
```

```
                socket.getOutputStream())),true);
        for(int i = 0; i < 10; i ++) {
          salida.println("Hola " + i);
          String str = entrada.readLine();
          System.out.println(str);
        }
        salida.println("FIN");
      } finally {
        System.out.println("cerrando...");
        socket.close();
      }
    }
} ///:~
```

En el método **main( )** se pueden ver las tres formas de producir la **InetAddress** de la dirección IP del bucle local: utilizando **null**, **localhost** o la dirección explícitamente reservada **127.0.0.1**. Por supuesto, si se desea establecer una conexión con una máquina que está en la red, deberá sustituirse esa dirección por la dirección IP de la máquina. Al imprimir la **InetAddress addr** (vía la llamada automática a su método **toString( )**) el resultado es:

```
localhost/127.0.0.1
```

Pasando a **getByName( )** un **null**, éste encontraba por defecto el **localhost**, lo que producía la dirección especial **127.0.0.1**.

Nótese que el **Socket** de nombre **socket** se crea tanto con la **InetAddress** como con el número de puerto. Para entender lo que significa imprimir uno de estos objetos **Socket**, recuérdese que una conexión a Internet viene determinada exclusivamente por estos cuatro fragmentos de datos: **clientHost**, **clientPortNumber**, **serverHost** y **serverPortNumber**. Cuando aparece un servidor, toma su puerto asignado (8080) en el bucle local (127.0.0.1). Al aparecer el cliente, se le asigna el siguiente puerto disponible en la máquina, en este caso el 1077, que resulta estar en la misma máquina que el servidor. Ahora, para que los datos se muevan entre el cliente y el servidor, cada uno de los extremos debe saber a dónde enviarlos. Por consiguiente, durante el proceso de conexión al servidor "conocido", el cliente envía una "dirección de retorno" de forma que el servidor pueda saber a dónde enviar sus datos. Esto es lo que se aprecia en la salida ejemplo para el lado servidor:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Esto significa que el servidor acaba de aceptar una conexión de 127.0.0.1 en el puerto 1077 al escuchar en su puerto local (8080). En el lado cliente:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

lo que significa que el cliente hizo una conexión con la 127.0.0.1 en el puerto 8080 usando el puerto local 1077.

Se verá que cada vez que se arranca de nuevo el cliente, se incrementa el número de puerto local. Empieza en el 1025 (uno más del bloque de puertos reservados) y va creciendo hasta volver a arrancar la máquina, momento en el que vuelve a empezar en el 1025. (En las máquinas UNIX, una vez que se llega al límite superior del rango de sockets, los números vuelven también al número mínimo disponible.)

Una vez que se ha creado el objeto **Socket**, el proceso de convertirlo en un **BufferedReader** y en un **PrintWriter** es el mismo que en el servidor (de nuevo, en ambas ocasiones se empieza con un **Socket**). Aquí, el cliente inicia la conversación enviando la cadena "hola" seguida de un número. Nótese que es necesario volver a vaciar el espacio de almacenamiento intermedio (cosa que ocurre automáticamente vía el segundo argumento al constructor **PrintWriter**). Si no se vacía el espacio de almacenamiento intermedio, se colgaría toda la conversación, pues nunca se llegaría a enviar el primer "hola" (el espacio de almacenamiento intermedio no estaría lo suficientemente lleno como para que se produjera automáticamente un envío). Cada línea que se envíe de vuelta al servidor se escribe en **System.out** para verificar que todo está funcionando correctamente. Para terminar la conversación, se envía el "FIN" preacordado. Si el cliente simplemente cuelga, el servidor enviará una excepción.

Se puede ver que en el ejemplo se tiene el mismo cuidado para asegurar que los recursos de red representados por el **Socket** se limpien adecuadamente, mediante el uso de un bloque **try-finally**.

Los sockets producen una conexión "dedicada" que persiste hasta que sea desconectada explícitamente. (La conexión dedicada puede seguir siendo desconectada de forma no explícita si se cuelga alguno de los lados o intermediarios.) Esto significa que las dos partes están bloqueadas en la comunicación, estando la conexión constantemente abierta. Esto parece un enfoque lógico para las redes, pero añade algo de sobrecarga a la propia red. Más adelante, en este mismo capítulo se verá un enfoque distinto al funcionamiento en red, en el que las conexiones son únicamente temporales.

Servir a múltiples clientes

El **ServidorParlante** funciona, pero solamente puede manejar a un cliente en cada momento. En un servidor típico, se deseará poder tratar con muchos clientes simultáneamente. La respuesta es el multihilo, y en los lenguajes que no lo soportan directamente esto significa todo tipo de complicaciones. En el Capítulo 14 se vio que en Java el multihilo es tan simple como se pudo, considerando que es de por sí un aspecto complejo. Dado que la capacidad de usar hilos en Java es bastante directa, construir un servidor que gestione múltiples clientes es relativamente sencillo.

El esquema básico es construir un **ServerSocket** único en el servidor, e invocar a **accept( )** para que espere a una nueva conexión. Cuando **accept( )** devuelva un **Socket**, se toma éste y se usa para crear un nuevo hilo cuyo trabajo es servir a ese cliente en particular. Después, se llama de nuevo a **accept( )** para esperar a un nuevo cliente.

En el siguiente código servidor, puede verse que tiene una apariencia similar a la de **ServidorParlante.java**, excepto en que todas las operaciones que sirven a un cliente en particular han quedado desplazadas al interior de una clase hilo separada:

```
//: c15:ServidorMultiParlante.java
// Un servidor que usa el multihilo
// para manejar cualquier número de clientes.
import java.io.*;
import java.net.*;

class ServirUnParlante extends Thread {
  private Socket socket;
  private BufferedReader entrada;
  private PrintWriter salida;
  public servirUnParlante(Socket s)
      throws IOException {
    socket = s;
    entrada =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Habilitar el autovaciado:
    salida =
      new PrintWriter(
        new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())), true);
    // Si cualquiera de las llamadas de arriba lanza una
    // excepción, el llamador es responsable de
    // cerrar el socket. De lo contrario lo cerrara
    // el hilo.
    start(); // Llama a run()
  }
  public void run() {
    try {
      while (true) {
        String str = entrada.readLine();
        if (str.equals("FIN")) break;
        System.out.println("Haciendo eco: " + str);
        salida.println(str);
      }
      System.out.println("cerrando...");
    } catch(IOException e) {
      System.err.println("Excepcion E/S");
    } finally {
      try {
        socket.close();
      } catch(IOException e) {
        System.err.println("Socket sin cerrar");
```

```
            }
        }
    }
}

public class ServidorMultiParlante {
    static final int PUERTO = 8080;
    public static void main(String[] args)
            throws IOException {
        ServerSocket s = new ServerSocket(PUERTO);
        System.out.println("Servidor iniciado");
        try {
            while(true) {
                // Se bloquea hasta que se da una conexión:
                Socket socket = s.accept();
                try {
                    new ServirUnParlante(socket);
                } catch(IOException e) {
                    // Si falla, cerrar el socket,
                    // si no, lo cerrará el hilo:
                    socket.close();
                }
            }
        } finally {
            s.close();
        }
    }
} ///:~
```

El hilo **ServirUnParlante** toma el objeto **Socket** producido por **accept( )** en el método **main( )** cada vez que un cliente nuevo haga una conexión. Posteriormente, y como antes, crea un **BufferedReader** y un objeto **PrintWriter** con autovaciado utilizando el **Socket**. Finalmente, llama al método **start( )** especial de **Thread**, que lleva a cabo la inicialización de hilos e invoca después a **run( )**. Éste hace el mismo tipo de acción que en el ejemplo anterior: leer algo del socket y después reproducirlo de vuelta hasta leer la señal especial "FIN".

La responsabilidad de cara a limpiar el socket debe ser diseñada nuevamente con cuidado. En este caso, el socket se crea fuera de **ServirUnParlante**, por lo que es posible compartir esta responsabilidad. Si el constructor **ServirUnParlante** falla, simplemente lanzará la excepción al llamador, que a continuación limpiará el hilo. Pero si el constructor tiene éxito, será el objeto **ServirUnParlante** el que tome la responsabilidad de limpiar el hilo, en su **run( )**.

Nótese la simplicidad del **ServidorMultiParlante**. Como antes, se crea un **ServerSocket** y se invoca a **accept( )** para permitir una nueva conexión. Pero en esta ocasión, se pasa el valor de retorno de **accept( )** (un **Socket**) al constructor de **ServirUnParlante**, que crea un nuevo hilo para manejar esa conexión. Cuando acaba la conexión, el hilo simplemente desaparece.

Si falla la creación de **ServerSocket**, se lanza de nuevo la excepción a través del método **main( )**. Pero si la creación tiene éxito, el **try-finally** externo garantiza su limpieza. El **try-catch** interno simplemente previene del fallo del constructor de **ServirUnParlante**; si el constructor tiene éxito, el hilo **ServirUnParlante** cerrará el socket asociado.

Para probar que el servidor verdaderamente gestiona múltiples clientes, he aquí el siguiente programa, que crea muchos clientes (usando hilos) que se conectan al mismo servidor. El máximo número de hilos permitido viene determinado por **final int MAX_HILOS**.

```
//: c15:ClienteMultiParlante.java
// Cliente que prueba el ServidorMultiParlante
// arrancando múltiples clientes.
import java.net.*;
import java.io.*;

class HiloClienteParlante extends Thread {
  private Socket socket;
  private BufferedReader entrada;
  private PrintWriter salida;
  private static int contador = 0;
  private int id = contador++;
  private static int conteoHilos = 0;
  public static int conteoHilos() {
    return conteoHilos;
  }
  public HiloClienteParlante(InetAddress addr) {
    System.out.println("Construyendo el cliente " + id);
    conteoHilos++;
    try {
      socket =
        new Socket(addr, HiloClienteParlante.PUERTO);
    } catch(IOException e) {
      System.err.println("Fallo el Socket ");
      // Si falla la creación del socket,
      // no hay que limpiar nada.
    }
    try {
      entrada =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Habilitar el auto-vaciado:
      salida =
        new PrintWriter(
          new BufferedWriter(
```

```
                new OutputStreamWriter(
                   socket.getOutputStream())), true);
         start();
      } catch(IOException e) {
         // Debería cerrarse el socket al darse
         // cualquier otro fallo distinto del de
         // constructor:
         try {
            socket.close();
         } catch(IOException e2) {
            System.err.println("Socket sin cerrar");
         }
      }
      // De otra forma, el socket se cerrará en el
      // método run() del hilo.
   }
   public void run() {
      try {
         for(int i = 0; i < 25; i++) {
            salida.println("Cliente " + id + ": " + i);
            String str = entrada.readLine();
            System.out.println(str);
         }
         salida.println("FIN");
      } catch(IOException e) {
         System.err.println("Excepcion de E/S");
      } finally {
         // Cerrarlo siempre:
         try {
            socket.close();
         } catch(IOException e) {
            System.err.println("Socket sin cerrar");
         }
         conteoHilos--; // Acabando este hilo
      }
   }
}

public class ClienteMultiParlante {
   static final int MAX_HILOS = 40;
   public static void main(String[] args)
         throws IOException, InterruptedException {
      InetAddress addr =
         InetAddress.getByName(null);
      while(true) {
```

```
        if(HiloClienteParlante.conteoHilos()
            < MAX_HILOS)
          new HiloClienteParlante(addr);
        Thread.currentThread().sleep(100);
      }
    }
} ///:~
```

El constructor **HiloClienteParlante** toma un **InetAddress** y lo usa para abrir un **Socket**. Probablemente se empiece a ver el patrón: siempre se usa el **Socket** para crear algún tipo de objeto **Reader** y/o **Writer** (o **InputStream** y/u **OutputStream**), que es la única forma de usar el **Socket**. (Por supuesto, puede escribir una o dos clases que automaticen este proceso en vez de llevar a cabo todo el tecleado cuando éste se vuelve molesto.)De nuevo, **start( )** lleva a cabo la inicialización de hilos e invoca a **run( )**. Aquí, se envían los mensajes al servidor y se reproduce esa información proveniente del servidor en la pantalla. Sin embargo, el hilo tiene un tiempo de vida limitado y suele acabarse. Nótese que el socket se limpia si falla el constructor después de haber creado el socket, pero antes de que acabe el constructor. De otra forma, la responsabilidad de llamar al **close( )** para el socket se relega al método **run( )**.

El **conteoHilos** mantiene un seguimiento de cuántos objetos **HiloClienteParlante** existen en cada momento. Se incrementa como parte del constructor y se disminuye al acabar **run( )** (lo que significa que el hilo está finalizando). En **ClienteMultiParlante.main( )** puede verse que se prueba el número de hilos, y si hay demasiados no se crean más. Después, el método se duerme. De esta forma, algunos hilos acabarían eventualmente y se podrían crear más. Se puede experimentar con **MAX_HILOS** para ver dónde empieza a tener problemas un sistema en particular por la existencia de demasiadas conexiones.

Datagramas

Los ejemplos vistos hasta el momento usan el *Transmission Control Protocol* (TCP, conocido también como *sockets basados en flujos*), diseñado para garantizar la fiabilidad de manera que la información llegue siempre. Permite la retransmisión de datos perdidos, proporciona múltiples caminos a través de distintos enrutadores en caso de que uno falle, y los bytes se entregan en el mismo orden en que son enviados. Todo este control y fiabilidad tiene un coste: TCP supone una gran sobrecarga.

Hay un segundo protocolo, denominado *User Datagram Protocol* (UDP), que no garantiza que los paquetes se entreguen y que lleguen en el orden en que son enviados. Se llama "protocolo no orientado a la conexión" (TCP es un "protocolo orientado a la conexión"), lo cual no suena bien, pero dado que es muchísimo más rápido, en ocasiones es útil. Hay algunas aplicaciones, como una señal de audio, en las que no es crítico el hecho de que se puedan perder algunos paquetes, pero la velocidad es vital. O considérese un servidor de hora del día, en el que no importa verdaderamente el que se pierda algún mensaje. Además, algunas aplicaciones deberían ser capaces de disparar un mensaje UDP al servidor para asumir con posterioridad, si no hay respuesta en un periodo razonable de tiempo, que el mensaje se ha perdido.

Generalmente, se trabajará directamente programando bajo TCP, y sólo ocasionalmente se hará uso de UDP. Hay un tratamiento de UDP más completo, incluyendo un ejemplo, en la primera edición de este libro (disponible en el CD ROM adjunto a este libro, o como descarga gratuita de *http://www.BruceEckel.com*).

Utilizar URL en un applet

Es posible que un *applet* pueda mostrar cualquier URL a través del navegador web en el que se esté ejecutando. Esto se puede lograr con la línea:

```
getAppletContext().showDocument(u);
```

donde **u** es el objeto **URL**. He aquí un ejemplo simple que le remite a otra página web. Aunque simplemente se logra una redirección a una página HTML, también se podría remitir a la salida de un programa CGI.

```
//: c15:MostrarHTML.java
// <applet code=MostrarHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class MostrarHTML extends JApplet {
  JButton enviar = new JButton("Ir");
  JLabel l = new JLabel();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    enviar.addActionListener(new Al());
    cp.add(enviar);
    cp.add(l);
  }
  class Al implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      try {
        // Esto podría ser un programa CGI en vez de
        // una página HTML.
        URL u = new URL(getDocumentBase(),
          "MarcoBuscador.html");
        // Mostrar la salida de la URL usando
        // el navegador web, como una página ordinaria:
```

```
            getAppletContext().showDocument(u);
        } catch(Exception e) {
            l.setText(e.toString());
        }
      }
    }
    public static void main(String[] args) {
      Console.run(new MostrarHTML(), 100, 50);
    }
} ///:~
```

La belleza de la clase **URL** radica en cuánto hace por ti. Es posible conectarse a servidores web sin saber mucho o nada de lo que está ocurriendo verdaderamente.

Leer un archivo de un servidor

Una variación del programa anterior lee un archivo ubicado en el servidor. En este caso, el archivo es especificado por el cliente:

```
//: c15:Buscador.java
// <applet code=Buscador width=500 height=300>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Buscador extends JApplet {
  JButton buscarlo= new JButton("Coger los Datos");
  JTextField f =
    new JTextField("Buscador.java", 20);
  JTextArea t = new JTextArea(10,40);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    buscarlo.addActionListener(new BuscarL());
    cp.add(new JScrollPane(t));
    cp.add(f); cp.add(buscarlo);
  }
  public class BuscarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      try {
        URL url = new URL(getDocumentBase(),
          f.getText());
```

```
            t.setText(url + "\n");
            InputStream is = url.openStream();
            BufferedReader entrada = new BufferedReader(
              new InputStreamReader(is));
            String linea;
            while ((linea = entrada.readLine()) != null)
              t.append(linea + "\n");
          } catch(Exception ex) {
            t.append(ex.toString());
          }
        }
      }
      public static void main(String[] args) {
        Console.run(new Buscador(), 500, 300);
      }
    } ///:~
```

La creación del objeto **URL** es similar al ejemplo anterior —**getDocumentBase( )** es como antes el punto de comienzo, pero en esta ocasión, el nombre del archivo se lee de **JTextField**. Una vez que se crea el objeto **URL**, se ubica su versión **String** en el **JTextArea,** de forma que se pueda ver la apariencia que tiene. Después, se proporciona un **InputStream** desde el **URL**, que en este caso simplemente producirá un flujo de los caracteres del archivo. Después de convertir a un **Reader**, y del tratamiento por espacios de almacenamiento intermedio, se lee cada línea para ser añadida al **JTextArea**. Nótese que el **JTextArea** se ha ubicado dentro de un **JScrollPane,** de forma que todo el desplazamiento de pantallas se gestiona automáticamente.

Más aspectos de redes

Además de lo cubierto en este tratamiento introductorio hay, de hecho, bastantes más aspectos relacionados con las redes. La capacidad de red de Java también proporciona gran soporte para URL, incluyendo gestores de protocolos para distintos tipos de contenidos que pueden descubrirse en las distintos sitios Internet. Se pueden encontrar más facetas de tratamiento de red de Java descritas con gran detalle en *Java Network Programming*, de Elliote Rusty Harold (O'Reilly, 1997).

Conectividad a Bases de Datos de Java (JDBC)

Se ha estimado que la mitad de todos los desarrollos software involucran aplicaciones cliente/servidor. Una gran promesa de Java ha sido la habilidad de construir aplicaciones de base de datos cliente/servidor independientes de la plataforma. En realidad esto se ha conseguido gracias a la Conectividad a Bases de Datos de Java (JDBC, *Java DataBase Connectivity*).

Uno de los mayores problemas de las bases de datos ha sido la guerra de características entre las propias compañías de bases de datos. Hay un lenguaje de bases de datos "estándar", el *Structure Query Languaje* (SQL-92), pero probablemente todo el mundo conoce el proveedor de la base de datos con la que trabaja a pesar del estándar. JDBC está diseñado para ser independiente de la plataforma, por lo que en tiempo de programación no hay que preocuparse por la base de datos que se esté utilizando. Sin embargo, sigue siendo posible construir llamadas específicas de ese fabricante desde JDBC, por lo que no se puede decir que haya restricción alguna a la hora de hacer lo que se tenga hacer.

Un lugar en el que los programadores pueden necesitar usar nombres del tipo SQL es en la sentencia `TABLE CREATE` de SQL al crear una nueva tabla de base de datos y definir el tipo SQL de cada columna. Desgraciadamente hay bastantes variaciones entre los tipos de SQL soportados por los distintos productos de base de datos. Bases de datos distintas que soportan tipos SQL con la misma semántica y estructura puede que den a esos tipos distintos nombres. La mayoría de las bases de datos principales soportan un tipo de datos SQL para valores binarios grandes: en Oracle, a este tipo se le denomina `LONG RAW`, Sybase lo llama `IMAGE`, Informix lo llama `BYTE`, y DB2 le llama `LONG VARCHAR FOR BIT DATA`. Por consiguiente, si la portabilidad de la base de datos es una de las metas a lograr, debería intentarse utilizar exclusívamente identificadores de tipos SQL genéricos.

La portabilidad es importante al escribir para un libro en el que los lectores pueden estar probando los ejemplos con cualquier tipo de almacén de datos desconocido. Hemos intentado escribir estos ejemplos para que sean lo más portables posible. El lector también podría darse cuenta de que se ha aislado el código específico de la base de datos para centralizar cualquier cambio que haya que realizar para hacer que estos ejemplos sean operativos en algún otro entorno.

JDBC, como muchas de las API de Java, está diseñado persiguiendo la simplicidad. Las llamadas a métodos que se hagan se corresponden con las operaciones lógicas que uno pensaría que debe hacer al recopilar datos desde la base de datos: conectarse a la base de datos, crear una sentencia y ejecutar la petición, y tener acceso al conjunto resultado.

Para permitir esta independencia de la plataforma, JDBC proporciona un *gestor de controladores* que mantiene dinámicamente todos los objetos controlador que puedan requerir las consultas a la base de datos. Por tanto, si se tienen tres tipos distintos de bases de datos a las que se desea establecer comunicación, se necesitarán tres objetos controlador. Estos objetos se registran a sí mismos en el gestor de controladores en el momento de su carga, y se puede forzar esta carga utilizando **Class.forName( )**.

Para abrir una base de datos, hay que crear un "URL de base de datos" que especifica:

1. Que se está usando JDBC con "jdbc".
2. El "subprotocolo": el nombre del controlador o el nombre de un mecanismo de conectividad de base de datos. Puesto que el diseño de JDBC se inspiró en ODBC, el primer subprotocolo disponible es el puente "jdbc-odbc", denominado "odbc".
3. El identificador de la base de datos. Éste varía en función del controlador de base de datos que se use, pero generalmente proporciona un nombre lógico al que el software administrador de la base de datos vincula a un directorio físico en el que están almacenadas las tablas de la base

de datos. Para que un identificador de base de datos tenga algún significado hay que registrar el nombre utilizando el software de administración de base de datos. (Este proceso de registro varía de plataforma a plataforma.)

Toda esta información se combina en un String, el "URL de base de datos". Por ejemplo, para conectarse a través del subprotocolo ODBC a una base de datos denominada "gente", el URL de la base de datos podría ser:

```
String dbUrl = "jdbc:odbc:gente";
```

Si se está estableciendo una conexión a través de una red, el URL de la base de datos debe contener la información de conexión que identifique la máquina remota, pudiendo resultar algo intimidatorio. He aquí un ejemplo de una base de datos Fuga a la que se invoca desde un cliente remoto utilizando RMI:

```
jdbc:rmi://192.168.170.27:1099/jdbc:fuga:db
```

La URL de la base de datos es verdaderamente dos llamadas a jdbc en una. La primera parte, `"jdbc:rmi://192.168.170.27:1099/"` usa RMI para hacer la conexión al motor de base de datos remota escuchando en el puerto 1099 de la dirección IP 192.168.170.27. La segunda parte del URL, "`jdbc:fuga:db`" conlleva los ajustes más típicos al usar el subprotocolo y el nombre de la base de datos, pero esto sólo ocurrirá una vez que la primera sección haya hecho la conexión vía RMI a la máquina remota.

Cuando uno está listo para conectarse a la base datos, hay que invocar al método **static DriverManager.getConnection( )** y pasarle el URL de la base de datos, el nombre de usuario y la contraseña para entrar en la base de datos. A cambio, se recibe un objeto **Connection** que puede ser usado posteriormente para preguntar y manipular la base de datos.

El ejemplo siguiente abre una base de datos de información de contacto y busca el apellido de una persona, suministrado en línea de comandos. Sólo selecciona los nombres de la gente que tienen dirección de correo electrónico, y después imprime todos los que casen con el apellido suministrado:

```
//: c15:jdbc:Buscar.java
// Busca direcciones de correo electrónico
// en una base de datos local usando JDBC.
import java.sql.*;

public class Buscar {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException {
    String dbUrl = "jdbc:odbc:gente";
    String usuario = "";
    String contrasena = "";
    // Cargar el controlador (se registra solo)
    Class.forName(
      "sun.jdbc.odbc.JdbcOdbcDriver");
```

```
      Connection c = DriverManager.getConnection(
        dbUrl, usuario, contrasena);
      Statement s = c.createStatement();
      // Código SQL:
      ResultSet r =
        s.executeQuery(
          "SELECT PRIMERO, SEGUNDO, CORREO " +
          "FROM gente.csv gente " +
          "WHERE " +
          "(LAST='" + args[0] + "') " +
          " AND (correo Is Not Null) " +
          "ORDER BY FIRST");
      while(r.next()) {
        // El uso de mayúsculas no importa:
        System.out.println(
          r.getString("Ultimo") + ", "
          + r.getString("pRIMERO")
          + ": " + r.getString("CORREO") );
      }
      s.close(); // También cierra el Resultset
  }
} ///:~
```

Puede verse que la creación del URL de la base de datos se hace como se describió anteriormente. En este ejemplo, no hay protección por contraseñas en la base de datos, por lo que los campos nombre de usuario y contraseña son cadenas de caracteres vacías.

Una vez que se hace la conexión con **DriverManager.getConnection( )**, se puede usar el objeto **Connection** resultante para crear un objeto **Statement** utilizando el método **createStatement( )**. Con el **Statement** resultante, se puede llamar a **executeQuery( )**, pasándole una cadena de caracteres que contenga cualquier sentencia SQL compatible con el estándar SQL-92. (En breve se verá cómo se puede generar esta sentencia automáticamente, evitando tener que saber mucho SQL.)

El método **executeQuery( )** devuelve un objeto **ResultSet**, que es un iterador: el método **next( )** mueve el iterador al siguiente recurso de la sentencia, o devuelve **false** si se ha alcanzado el fin del conjunto resultado. Una ejecución de **executeQuery( )** siempre genera un objeto **ResultSet** incluso si la solicitud conlleva un conjunto vacío (es decir, no se lanza una excepción). Nótese que hay que llamar a **next( )** una vez, por lo menos antes de intentar leer ningún registro. Si el conjunto resultado está vacío, esta primera llamada a **next( )** devolverá **false**. Por cada registro del conjunto resultado, es posible seleccionar los campos usando (entre otros enfoques) el nombre del campo como cadena de caracteres. Nótese también que se ignora el uso de mayúsculas o minúsculas en el nombre del campos —con una base de datos SQL no importa. El tipo de dato que se devuelve se determina invocando a **getInt( )**, **getString( )**, **getFloat( )**, etc. En este momento, se tienen los datos de la base de datos en formato Java nativo y se puede hacer lo que se desee con ellos usando código Java ordinario.

Hacer que el ejemplo funcione

Con JDBC, entender el código es relativamente fácil. La parte complicada es hacer que funcione en un sistema en particular. La razón por la que es complicada es que requiere que cada lector averigüe cómo cargar adecuadamente su controlador JDBC, y cómo configurar una base de datos utilizando su software de administración de base de datos. Por supuesto, este proceso puede variar radicalmente de máquina a máquina, pero el proceso que sabíamos usar para que funcionara bajo un Windows de 32 bits puede dar algunas pistas para que cada uno se enfrente a su propia situación:

Paso 1: Encontrar el Controlador JDBC

El programa de arriba contiene la sentencia:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Ésta implica una estructura de directorios que es determinante. Con esta instalación particular del JDK 1.1, no había ningún archivo llamado **JdbcOdbcDriver.class**, por lo que si se echa un vistazo a este ejemplo y se empieza a buscarlo, uno se frustra. Otros ejemplos publicados usan un pseudonombre, como "miControlador.ClassName", que es aún menos útil. De hecho, la sentencia de carga de arriba para el controlador jdbc-odbc (el único que, de hecho, viene con el JDK) sólo aparece en unos pocos sitios en la documentación en línea (en particular, en una página de nombre "JDBC-ODBC Bridge Driver"). Si la sentencia de carga de arriba no funciona, entonces puede que el nombre haya cambiado como parte de los cambios de versión de Java, por lo que habría que volver a recorrerse toda la documentación.

Si la sentencia de carga está mal, se obtendrá justo en ese momento una excepción. Para probar si la sentencia de carga del controlador está funcionando correctamente o no, puede marcarse como comentario la sentencia; si el programa no lanza excepciones, será señal de que el controlador se está cargando correctamente.

Paso 2: Configurar la base de datos

Esto es, de nuevo, específico al Windows de 32 bits; puede ser que alguien tenga que investigar si está trabajando con otra plataforma.

En primer lugar, abra el panel de control. Se verá que hay dos iconos que dicen "ODBC". Hay que usar el que dice "Fuentes de Datos ODBC 32 bits", puesto que el otro es para lograr retrocompatibilidad con software ODBC de 16 bits y no generará ningún resultado en el caso de JDBC. Cuando se abre el icono "Fuentes de Datos ODBC 32 bits", se verá una caja de diálogo con solapas, entre las que se incluyen "DSN de Usuario", "DSN de Sistema", "DSN de Archivo", etc., donde DSN significa "*Data Source Name*" ("Nombre de la fuente de datos"). Resulta que para el puente JDBC-ODBC, el único lugar en el que es importante configurar la base de datos es "DSN del Sistema", pero también se deseará probar la configuración y hacer consultas, y para ello también será necesario configurar la base de datos en "DSN de Archivo". Esto permitirá a la herramienta Microsoft Query (que viene con Microsoft Office) encontrar la base de datos. Nótese que existen también otras herramientas de otros vendedores.

La base de datos más interesante es una que esté actualmente en uso. El ODBC estándar soporta varios formatos de archivos distintos, incluyendo algunos tan venerables como DBase. Sin embargo, también incluye el formato simple "ASCII separado por comas", que generalmente toda herramienta de datos tiene la capacidad de escribir. Simplemente tomamos la base de datos de "gente", mantenida por nosotros durante años usando herramientas de gestión de contactos, y la exportamos a un archivo ASCII separado por comas (que suelen tener extensión **.csv**). En la sección "DSN de sistema", seleccionamos "Agregar", elegimos el controlador de texto para gestionar nuestro archivo ASCII separado por comas, y después deseleccionamos "usar directorio actual" para que me pemitiera especificar el directorio al que exportar el archivo de datos.

Se verá al hacer esto que verdaderamente no se especifica un archivo, sólo un directorio. Eso es porque una base de datos suele representarse como una colección de archivos bajo un único directorio (aunque podría estar representada también de otra forma). Cada archivo suele contener una única tabla, y las sentencias SQL pueden producir resultados provenientes de múltiples tablas de la base de datos (a esto se le llama *join*). A una base de datos que sólo contiene una base de datos (como mi base de datos "gente") se le suele llamar *flat-file database*. La mayoría de problemas que van más allá del simple almacenamiento y recuperación de datos suelen requerir de varias tablas, que deben estar relacionadas mediante *joins* para producir los resultados deseados, y a éstas se las denomina bases de datos *relacionales*.

Paso 3: Probar la configuración

Para probar la configuración, será necesario disponer de alguna forma de descubrir si la base de datos es visible desde un programa que hace consultas a la misma. Por supuesto, se puede ejecutar simplemente el programa ejemplo JDBC de arriba, incluyendo la sentencia:

```
Connection c = DriverManager.getConnection(
  dbUrl, usuario, contrasena);
```

Si se lanza una excepción, la configuración era incorrecta.

Sin embargo, es útil hacer que una herramienta de generación de consultas se vea involucrada en esto. Utilizamos Microsoft Query, que viene con Microsoft Office, pero podría preferirse alguna otra. La herramienta de consultas debería saber dónde está la base de datos y Microsoft Query exigía que fuera al campo Administrador de ODBC de la etiqueta "DSN de Archivo" y añadiera una nueva entrada ahí, especificando de nuevo el controlador de texto y el directorio en el que reside nuestra base de datos. Se puede nombrar a la entrada como se desee, pero es útil usar el mismo nombre usado en "DSN de Sistema".

Una vez hecho esto se verá que la base de datos está disponible al crear una nueva consulta usando la herramienta de consultas.

Paso 4: Generar la consulta SQL

La consulta que creamos usando Microsoft Query no sólo nos mostraba que la base de datos estaba ahí y en orden correcto, sino que también creaba automáticamente el código SQL que necesitaba insertar en nuestro programa Java. Queríamos una consulta que buscara los registros que tuvie-

ran el mismo apellido que se tecleara en la línea de comandos al arrancar el programa Java. Por tanto, y como punto de partida, buscamos un apellido específico, "Eckel". También queríamos que se mostraran sólo aquellos nombres que tuvieran alguna dirección de correo electrónico asociada. Los pasos que seguimos para crear esta consulta fueron:

1. Empezar una nueva consulta y utilizar el Query Wizard. Seleccionar la base de datos "gente". (Esto equivale a abrir la conexión de base de datos usando el URL de base de datos apropiado).

2. Seleccionar la tabla "gente" dentro de la base de datos. Desde dentro de la tabla, seleccionar las columnas PRIMERO, SEGUNDO y CORREO.

3. Bajo "Filtro de Datos", seleccionar SEGUNDO y elegir "equals" con un argumento "Eckel". Hacer clic en el botón de opción "And".

4. Seleccionar CORREO y elegir "Is not Null".

5. Bajo "Sort by", seleccionar PRIMERO.

Los resultados de esta consulta mostrarán si se está obteniendo lo deseado.

Ahora se puede presionar el botón SQL y sin hacer ningún tipo de investigación, aparecerá el código SQL correcto, listo para ser cortado y pegado. Para esta consulta, sería algo así:

```
SELECT gente.PRIMERO, gente.SEGUNDO, gente.CORREO
FROM gente.csv gente
WHERE (gente.SEGUNDO='Eckel') AND
(gente.CORREO Is Not Null)
ORDER BY gente.PRIMERO
```

Es posible que las cosas vayan mal, especialmente si las consultas son más complicadas, pero usando una herramienta de generación de consultas, se pueden probar éstas de forma interactiva y generar automáticamente el código correcto. Es difícil defender la postura de hacer esto a mano.

Paso 5: Modificar y cortar en una consulta

Se verá que el código de arriba tiene distinto aspecto del que se usó en el programa. Eso es porque la herramienta de consultas usa especificación completa de todos los nombres, incluso cuando sólo esté involucrada una tabla. (Cuando hay más de una tabla, la especificación completa evita colisiones entre columnas de distintas tablas que pudieran tener igual nombre.) Puesto que esta consulta simplemente involucra a una tabla, se puede eliminar el especificador "gente" de la mayoría de los nombres, así:

```
SELECT PRIMERO,SEGUNDO,CORREO
FROM gente.csv gente
WHERE (SEGUNDO='Eckel') AND
(EMAIL Is Not Null)
ORDER BY PRIMERO
```

Además, no se deseará codificar todo el programa para que sólo busque un nombre en concreto. En vez de ello, se debería buscar el nombre proporcionado como parámetro de línea de comandos. Hacer estos cambios y convertir la sentencia SQL en un **String** de creación dinámica produce:

```
"SELECT PRIMERO,SEGUNDO CORREO " +
"FROM gente.csv gente " +
"WHERE " +
" (LAST='" + args[0] + "') " +
" AND (CORREO Is Not Null) " +
"ORDER BY PRIMERO");
```

SQL tiene otra forma de insertar nombres en una consulta, denominado *procedimientos almacenados*, que se usan para lograr incrementos de velocidad. Pero como experimentación de base de datos para un primer enfoque, construir las cadenas de consulta en Java es una opción más que correcta.

A partir de este ejemplo se puede ver que usando las herramientas actualmente disponibles —en particular la herramienta de construcción de consultas— es bastante directo programar con SQL y JDBC.

Una versión con IGU del programa de búsqueda

Es más útil dejar que el programa de búsqueda se ejecute continuamente y simplemente cambiar al mismo y teclear un nombre cuando se desee buscar a alguien. El siguiente programa crea el programa de búsqueda como una aplicación/*applet* y también añade finalización automática, de forma que los datos se mostrarán sin forzar al lector a teclear el apellido completo:

```
//: c15:jdbc:BuscarV.java
//versión IGU de Buscar.java.
// <applet code=BuscarV
// width=500 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.sql.*;
import com.bruceeckel.swing.*;

public class BuscarV extends JApplet {
  String dbUrl = "jdbc:odbc:gente";
  String usuario = "";
  String contrasena = "";
  Statement s;
  JTextField buscarPor = new JTextField(20);
  JLabel conclusion =
    new JLabel("                    ");
```

```
  JTextArea resultados = new JTextArea(40, 20);
  public void init() {
    buscarPor.getDocument().addDocumentListener(
      new BuscarL());
    JPanel p = new JPanel();
    p.add(new Label("Apellido a buscar:"));
    p.add(buscarPor);
    p.add(conclusion);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    cp.add(resultados, BorderLayout.CENTER);
    try {
      // Cargar el controlador (se registra automáticamente)
      Class.forName(
        "sun.jdbc.odbc.JdbcOdbcDriver");
      Connection c = DriverManager.getConnection(
        dbUrl, usuario, contrasena);
      s = c.createStatement();
    } catch(Exception e) {
      resultados.setText(e.toString());
    }
  }
  class BuscarL implements DocumentListener {
    public void changedUpdate(DocumentEvent e){}
    public void insertUpdate(DocumentEvent e){
      valorTextoCambiado();
    }
    public void removeUpdate(DocumentEvent e){
      valorTextoCambiado();
    }
  }
  public void valorTextoCambiado() {
    ResultSet r;
    if(buscarPor.getText().length() == 0) {
      conclusion.setText("");
      resultados.setText("");
      return;
    }
    try {
      // Finalización de nombres:
      r = s.executeQuery(
        "SELECT SEGUNDO FROM gente.csv gente " +
        "WHERE (SEGUNDO Like '" +
        buscarPor.getText()  +
        "%') ORDER BY SEGUNDO");
```

```
      if(r.next())
        conclusion.setText(
          r.getString("segundo"));
      r = s.executeQuery(
        "SELECT PRIMERO, SEGUNDO, CORREO " +
        "FROM gente.csv gente " +
        "WHERE (SEGUNDO='" +
        conclusion.getText() +
        "') AND (CORREO Is Not Null) " +
        "ORDER BY PRIMERO");
    } catch(Exception e) {
      resultados.setText(
        buscarPor.getText() + "\n");
      resultados.append(e.toString());
      return;
    }
    resultados.setText("");
    try {
      while(r.next()) {
        resultados.append(
          r.getString("Primero") + ", "
          + r.getString("sEGUNDO") +
          ": " + r.getString("CORREO") + "\n");
      }
    } catch(Exception e) {
      resultados.setText(e.toString());
    }
  }
  public static void main(String[] args) {
    Console.run(new BuscarV(), 500, 200);
  }
} ///:~
```

Mucha de la lógica de la base de datos es la misma, pero puede verse que se añade un **DocumentListener** al **JTextField** (véase la entrada **javax.swing.JTextField** de la documentación Java HTML de *http://www.java.sun.com* para encontrar detalles), de forma que siempre que se teclee un nuevo carácter, primero se intenta terminar el apellido buscándolo en la base de datos y usando el primero que se muestre. (Se coloca en la **JLabel conclusion**, y se usa como texto de búsqueda). De esta forma, tan pronto como se hayan tecleado los suficientes caracteres para que el programa encuentre el apellido buscado de manera unívoca, se puede parar el mismo.

Por qué el API JDBC parece tan complejo

Cuando se accede a la documentación en línea de JDBC puede asustar. En concreto, en la interfaz **DatabaseMetaData** —que es simplemente vasta, contrariamente al resto de interfaces de Java— hay mé-

todos como **dataDefinitionCausesTransactionCommit( )**, **getMaxColumnNameLength( )**, **getMaxStatementLength( )**, **storesMixedCaseQuotedIdentifiers( )**, **supportsANSI92IntermediateSQL( )**, **supportsLimitedOuterJoins( )** y demás. ¿De qué va todo esto?

Como se mencionó anteriormente, las bases de datos, desde su creación, siempre han parecido una fuente constante de tumultos, especialmente debido a la demanda de aplicaciones de bases de datos, y por consiguiente, las herramientas de bases de datos suelen ser tremendamente grandes. Recientemente —y sólo recientemente— ha habido una convergencia en el lenguaje común SQL (y hay muchos otros lenguajes comunes de bases de datos de uso frecuente). Pero incluso con un SQL "estándar" hay tantas variaciones del tema que JDBC debe proporcionar la gran interfaz **DatabaseMetaData,** de forma que el código pueda descubrir las capacidades del SQL estándar en particular que usa la base de datos a la que se está actualmente conectado. Abreviadamente, puede escribirse SQL simple y transportable, pero si se desea optimizar la velocidad, la codificación se multiplicará tremendamente al investigar las capacidades de la base de datos de un vendedor concreto.

Esto, por supuesto, no es culpa de Java. Las discrepancias entre los productos de base de datos son simplemente algo que JDBC intenta ayudar a compensar. Pero recuerde que le puede facilitar las cosas la escritura de sentencias de base de datos *(queries)* genéricas, sin apenas preocupar al rendimiento, o, si debe afinar en el rendimiento, conozca la plataforma en la que está escribiendo para evitar tener que escribir todo ese código de investigación.

Un ejemplo más sofisticado

Un segundo ejemplo[2] más interesante involucra una base de datos multitabla que reside en un servidor. Aquí, la base de datos está destinada a proporcionar un repositorio de actividades de la comunidad y permitir a la gente apuntarse a esos eventos, de forma que se le llama Base de Datos de Interés de la Comunidad (CID, o *Community Interests Database*). Este ejemplo sólo proporcionará un repaso de la base de datos y su implementación. Hay muchos libros, seminarios y paquetes software que nos ayudarán a diseñar y desarrollar una base de datos.

Además, este ejemplo presupone que anteriormente se ha instalado una base de datos SQL en un servidor (aunque también podría ejecutarse en una maquina local), y que se ha integrado y descubierto un controlador JDBC apropiado para esa base de datos. Existen varias bases de datos SQL gratuitas disponibles, y algunas se instalan incluso automáticamente con varias versiones de Linux. Cada una es responsable de elegir la base de datos y de ubicar el controlador JDBC; este ejemplo está basado en una base de datos SQL llamada "Fuga".

Para facilitar los cambios en la información de conexión, el controlador de la base de datos, el URL de la misma, el nombre de usuario y la contraseña se colocaron en una clase diferente:

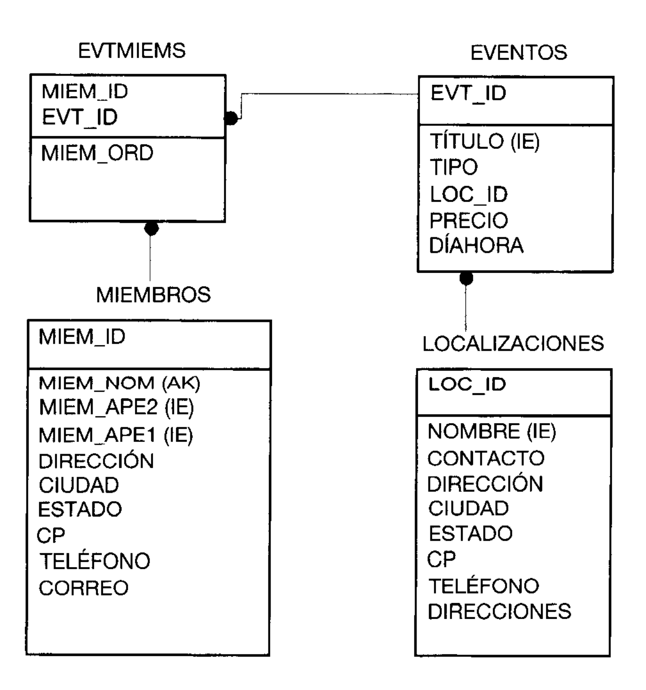
```
//: c15:jdbc:ConectarCID.java
// Información de conexión a base de datos para
// la base de datos de interés de la comunidad (CID).
```

[2] Creado por Dave Bartlett.

```
public class ConectarCID {
  // Toda la información específica de Fuga:
  public static String controladorBD =
    "COM.fuga.core.JDBCDriver";
  public static String URLdb =
    "jdbc:fuga:d:/docs/_work/JSapienDB";
  public static String usuario = "";
  public static String contrasena = "";
} ///:~
```

En este ejemplo, no hay protección por contraseñas en la base de datos, por lo que el nombre de usuario y la contraseña están vacíos.

La base de datos consiste en un conjunto de tablas con la estructura que se muestra:



"Miembros" contiene la información de miembros de la comunidad, "Eventos" y "Localizaciones" contienen información sobre las actividades y cuándo tendrán lugar, y "Evtmiems" conecta los eventos con los miembros a los que les gustaría asistir. Puede verse que un miembro de datos de una tabla produce una clave en la otra.

La clase siguiente contiene las cadenas SQL que crearán estas tablas de bases de datos (véase una guía de SQL si se desea obtener explicación del código SQL):

```
//: c15:jdbc:CIDSQL.java
// Strings SQL para crear las tablas de la CID.
```

```
public class CIDSQL {
  public static String[] sql = {
    // Crear la tabla MIEMBROS:
    "drop table MIEMBROS",
    "create table MIEMBROS " +
    "(MIEM_ID INTEGER primary key, " +
    "MIEM_NOM VARCHAR(12) not null unique, "+
    "MIEM_APE2 VARCHAR(40), " +
    "MIEM_APE1 VARCHAR(20), " +
    "DIRECCION VARCHAR(40), " +
    "CIUDAD VARCHAR(20), " +
    "ESTADO CHAR(4), " +
    "CP CHAR(5), " +
    "TELEFONO CHAR(12), " +
    "CORREO VARCHAR(30))",
    "create unique index " +
    "APE2_IND on MIEMBROS(MIEM_APE2)",
    // Crear la tabla EVENTOS
    "drop table EVENTOS",
    "create table EVENTOS " +
    "(EVT_ID INTEGER primary key, " +
    "EVT_TITULO VARCHAR(30) not null, " +
    "EVT_TIPO VARCHAR(20), " +
    "LOC_ID INTEGER, " +
    "PRECIO DECIMAL, " +
    "DIAHORA TIMESTAMP)",
    "create unique index " +
    "TITULO_IND on EVENTOS(EVT_TITULO)",
    // Crear la tabla EVTMIEMS
    "drop table EVTMIEMS",
    "create table EVTMIEMS " +
    "(MIEM_ID INTEGER not null, " +
    "EVT_ID INTEGER not null, " +
    "MIEM_ORD INTEGER)",
    "create unique index " +
    "EVTMIEM_IND on EVTMIEMS(MIEM_ID, EVT_ID)",
    // Crear la tabla LOCALIZACIONES
    "drop table LOCALIZACIONES",
    "create table LOCALIZACIONES " +
    "(LOC_ID INTEGER primary key, " +
    "LOC_NOMBRE VARCHAR(30) not null, " +
    "CONTACTO VARCHAR(50), " +
    "DIRECCION VARCHAR(40), " +
    "CIUDAD VARCHAR(20), " +
    "ESTADO VARCHAR(4), " +
```

```
        "CP VARCHAR(5), " +
        "TELEFONO CHAR(12), " +
        "DIRECCIONES VARCHAR(4096))",
        "create unique index " +
        "NOMBRE_IND on LOCALIZACIONES(LOC_NOMBRE)",
    };
} ///:~
```

El programa siguiente usa la información de **ConectarCID** y **CIDSQL** para cargar el controlador JDBC y establecer la conexión a la base de datos y crear después la estructura de tablas del diagrama de arriba. Para conectar con la base de datos se invoca al método **static DriverManager.getConnection( )**, pasándole el URL de la base de datos, el nombre de usuario y una contraseña para entrar en la misma. Al retornar, se obtiene un objeto **Connection** que puede usarse para hacer consultas y manipular la base de datos. Una vez establecida la conexión se puede simplemente meter el SQL en la base de datos, en este caso, recorriendo el array **CIDSQL**. Sin embargo, la primera vez que se ejecute el programa, el comando "drop table" fallará, causando una excepción, que es capturada, y de la que se informa, para finalmente ignorarla. La razón del comando "drop table" es permitir una experimentación sencilla: se puede modificar el SQL que define las tablas y después volver a ejecutar el programa, lo que hará que las viejas tablas sean reemplazadas por las nuevas.

En este ejemplo, tiene sentido dejar que se lancen las excepciones a la consola:

```
//: c15:jdbc:CrearTablasCID.java
// Crea las tablas de base de datos
// para la base de datos de interés de la comunidad.
import java.sql.*;

public class CrearTablasCID {
    public static void main(String[] args)
    throws SQLException, ClassNotFoundException,
    IllegalAccessException {
        // Cargar el controlador (se registra a sí mismo)
        Class.forName(ConectarCID.controladorBD);
        Connection c = DriverManager.getConnection(
            ConectarCID.URLdb, ConectarCID.usuario,
            ConectarCID.contrasena);
        Statement s = c.createStatement();
        for(int i = 0; i < CIDSQL.sql.length; i++) {
            System.out.println(CIDSQL.sql[i]);
            try {
                s.executeUpdate(CIDSQL.sql[i]);
            } catch(SQLException sqlEx) {
                System.err.println(
                    "Probablemente falló un 'drop table' ");
            }
        }
```

```
    s.close();
    c.close();
  }
} ///:~
```

Nótese que se pueden controlar todos los cambios en la base de datos, cambiando **String**s en la tabla **CIDSQL**, sin modificar **CrearTablasCID**.

El método **executeUpdate( )** devolverá generalmente el número de las filas afectadas por la sentencia SQL. Este método se usa más frecuentemente para ejecutar sentencias `INSERT`, `UPDATE` o `DELETE`, para modificar una o más columnas. Para sentencias como `CREATE TABLE`, `DROP TABLE`, y `CREATE INDEX`, **executeUpdate( )** siempre devuelve cero.

Para probar la base de datos, se carga con algunos datos de ejemplo. Esto requiere una serie de `INSERT`s seguidos de una `SELECT` para producir un resultado conjunto. Para facilitar las adiciones y cambios a los datos de prueba, éste se dispone en un array bidimensional de **Object**s, y el método **executeInsert( )** puede usar después la información de una columna de la tabla para crear el comando SQL apropiado.

```
//: c15:jdbc:CargarBD.java
// Carga y prueba la base de datos.
import java.sql.*;

class PruebaConjunto {
  Object[][] datos = {
    { "MIEMBROS", new Integer(1),
      "dbartlett", "Bartlett", "David",
      "123 Mockingbird Lane",
      "Gettysburg", "PA", "19312",
      "123.456.7890",  "bart@you.net" },
    { "MIEMBROS", new Integer(2),
      "beckel", "Eckel", "Bruce",
      "123 Over Rainbow Lane",
      "Crested Butte", "CO", "81224",
      "123.456.7890", "beckel@you.net" },
    { "MIEMBROS", new Integer(3),
      "rcastaneda", "Castaneda", "Robert",
      "123 Downunder Lane",
      "Sydney", "NSW", "12345",
      "123.456.7890", "rcastaneda@you.net" },
    { "LOCALIZACIONES", new Integer(1),
      "Center for Arts",
      "Betty Wright", "123 Elk Ave.",
      "Crested Butte", "CO", "81224",
      "123.456.7890",
      "Ir de esta manera." },
```

```
    { "LOCALIZACIONES", new Integer(2),
      "Witts End Conference Center",
      "John Wittig", "123 Music Drive",
      "Zoneville", "PA", "19123",
      "123.456.7890",
      "Ir de esta manera." },
    { "EVENTOS", new Integer(1),
      "Project Management Myths",
      "Software Development",
      new Integer(1), new Float(2.50),
      "2000-07-17 19:30:00" },
    { "EVENTOS", new Integer(2),
      "Life of the Crested Dog",
      "Archeology",
      new Integer(2), new Float(0.00),
      "2000-07-19 19:00:00" },
    // Asociar gente a eventos
    {  "EVTMIEMS",
      new Integer(1),  // Dave va al evento
      new Integer(1),  // Software.
      new Integer(0) },
    { "EVTMIEMS",
      new Integer(2),  // Bruce va al evento
      new Integer(2),  // Arqueología.
      new Integer(0) },
    { "EVTMIEMS",
      new Integer(3),  // Robert va al evento
      new Integer(1),  // Software.
      new Integer(1) },
    { "EVTMIEMS",
      new Integer(3), // ... y
      new Integer(2), // al evento Arqueología.
      new Integer(1) },
  };
  // Usar el conjunto de datos por defecto:
  public PruebaConjunto() {}
  // Usar un conjunto de datos distinto:
  public PruebaConjunto(Object[][] dat) { datos = dat; }
}

public class CargarDB {
  Statement sentencia;
  Connection conexion;
  PruebaConjunto pconjunto;
  public CargarBD(PruebaConjunto p) throws SQLException {
```

```
    pconjunto = p;
    try {
      // Cargar el controlador (se registra solo)
      Class.forName(ConectarCID.controladorBD);
    } catch(java.lang.ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    conexion = DriverManager.getConnection(
      ConectarCID.URLdb, ConectarCID.usuario,
      ConectarCID.contrasena);
    sentencia = conexion.createStatement();
  }
  public void limpiar() throws SQLException {
    sentencia.close();
    conexion.close();
  }
  public void ejecutarInsertar(Object[] datos) {
    String sql = "insert into "
      + datos[0] + " values(";
    for(int i = 1; i < datos.length; i++) {
      if(datos[i] instanceof String)
        sql += "'" + datos[i] + "'";
      else
        sql += datos[i];
      if(i < datos.length - 1)
        sql += ", ";
    }
    sql += ')';
    System.out.println(sql);
    try {
      sentencia.executeUpdate(sql);
    } catch(SQLException sqlEx) {
      System.err.println("Falló inserción.");
      while (sqlEx != null) {
        System.err.println(sqlEx.toString());
        sqlEx = sqlEx.getNextException();
      }
    }
  }
  public void cargar() {
    for(int i = 0; i< pconjunto.datos.length; i++)
      ejecutarInsertar(pconjunto.datos[i]);
  }
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
```

```
  throws SQLException {
    CargarBD db = new CargarBD(new PruebaConjunto());
    bd.cargar();
    try {
      // Obtener un ResultSet a partir de la base de datos cargada:
      ResultSet rs = db.statement.executeQuery(
        "select " +
        "e.EVT_TITULO, m.MIEM_APE2, m.MIEM_APE1 "+
        "from EVENTOS e, MIEMBROS m, EVTMIEMS em " +
        "where em.EVT_ID = 2 " +
        "and e.EVT_ID = em.EVT_ID " +
        "and m.MIEM_ID = em.MIEM_ID");
      while (rs.next())
        System.out.println(
          rs.getString(1) + "  " +
          rs.getString(2) + ", " +
          rs.getString(3));
    } finally {
      db.limpiar();
    }
  }
} ///:~
```

La clase **PruebaConjunto** contiene un conjunto de datos por defecto producido al usar el constructor por defecto; sin embargo, también se puede crear un objeto **PruebaConjunto** usando un conjunto de datos alternativo con el segundo constructor. El conjunto de datos se guarda en un array bidimensional de **Object** porque puede ser de cualquier tipo, incluidos **Strings** o tipos numéricos. El método **ejecutarInsertar( )** utiliza RTTI para distinguir entre datos **String** (que deben ir entre comillas)y los no-**String** a medida que construye el comando SQL a partir de los datos. Tras imprimir este programa en la consola, se usa **executeUpdate( )** para enviarlo a la base de datos.

El constructor de **CargarBD** hace la conexión, y **cargar( )** recorre los datos y llama a **ejecutarInsertar( )** por cada registro. El método **limpiar( )** cierra la sentencia y la conexión; para garantizar que se invoque, éste se ubica dentro de una cláusula finally.

Una vez cargada la base de datos, una sentencia **executeQuery( )** produce el resultado conjunto de ejemplo. Puesto que la consulta combina varias tablas, es un ejemplo de un *join*.

Hay más información de JDBC disponible en los documentos electrónicos que vienen como parte de la distribución de Java de Sun. Además, se puede encontrar más en el libro *JDBC Database Access with Java* (Hamilton, Cattel, y Fisher, Addison-Wesley, 1997). También suelen aparecer otros libros sobre este tema con bastante frecuencia.

Servlets

El acceso de clientes desde Internet o intranets corporativas es una forma segura de permitir a varios usuarios acceder a datos y recursos de forma sencilla[3]. Este tipo de acceso está basado en el uso de los estándares *Hypertext Markup Language* (HTML) e *Hypertext Transfer Protocol* (HTTP) de la World Wide Web por parte de los clientes. El conjunto de API Servlet abstrae un marco de solución común para responder a peticiones HTTP.

Tradicionalmente, la forma de gestionar un problema como el de permitir a un cliente de Internet actualizar una base de datos es crear una página HTML con campos de texto y un botón de "enviar". El usuario teclea la información apropiada en los campos de texto y presiona el botón "enviar". Los datos se envían junto con una URL que dice al servidor qué hacer con los datos especificando la ubicación de un programa *Common Gateway Interface* (CGI) que ejecuta el servidor, proporcionando al programa los datos al ser invocado. El programa CGI suele estar escrito en Perl, Python, C, C++ o cualquier lenguaje que pueda leer de la entrada estándar y escribir en la salida estándar. Esto es todo lo que es proporcionado por el servidor web: se invoca al programa CGI, y se usan flujos estándar (u, opcionalmente en caso de entrada, una variable de entorno) para la entrada y la salida. El programa CGI es responsable de todo lo demás. Primero, mira a los datos y decide si el formato es correcto. Si no, el programa CGI debe producir HTML para describir el problema; esta página se pasa al servidor Web (vía salida estándar del programa CGI), que lo vuelve a enviar al usuario. El usuario debe generalmente salvar la página e intentarlo de nuevo. Si los datos son correctos, el programa CGI los procesa de forma adecuada, añadiéndolos quizás a una base de datos. Después, debe producir una página HTML apropiada para que el servidor web se la devuelva al usuario.

Sería ideal ir a una solución completamente basada en Java para este ejemplo —un *applet* en el lado cliente que valide y envíe los datos, y un servlet en el lado servidor para recibir y procesar los datos. Desgraciadamente, aunque se ha demostrado que los *applets* son una tecnología con gran soporte, su uso en la Web ha sido problemático porque no se puede confiar en que en un navegador cliente web haya una versión particular de Java disponible; de hecho, ¡ni siquiera se puede confiar en que un navegador web soporte Java! En una Intranet, se puede requerir que esté disponible cierto soporte, lo que permite mucha mayor flexibilidad en lo que se puede hacer, pero en la Web el enfoque más seguro es manejar todo el proceso en el lado servidor y entregar HTML plano al cliente. De esa forma, no se negará a ningún cliente el uso del sitio porque no tenga el software apropiado instalado.

Dado que los servlets proporcionan una solución excelente para soporte en el lado servidor, son una de las razones más populares para migrar a Java. No sólo proporcionan un marco que sustituye a la programación CGI (y elimina muchos problemas complicados de los CGIs), sino que todo el código gana portabilidad de plataforma gracias al uso de Java, teniendo acceso a todos los APIs de Java (excepto, por supuesto, a los que producen IGU, como Swing).

---

[3] Dave Bartlett contribuyó desarrollo de este material, y también en la sección JSP.

El servlet básico

La arquitectura del API servlet es la de un proveedor de servicios clásico con un método **service( )** a través del cual se enviarán todas las peticiones del cliente por parte del software contenedor del servlet, y los métodos de ciclo de vida **init( )** y **destroy( )**, que se invocan sólo cuando se carga y descarga el servlet (esto raramente ocurre).

```
public interface Servlet
  public void init(ServletConfig config)
    throws ServletException;
  public ServletConfig getServletConfig();
  public void service(ServletRequest req,
    ServletResponse res)
    throws ServletException, IOException;
  public String getServletInfo();
  public void destroy();
}
```

El único propósito de **getServletConfig( )** es devolver un objeto **ServletConfig** que contenga los parámetros de inicialización y arranque de este servidor. El método **getServletInfo( )** devuelve una cadena de caracteres que contiene información sobre el servlet, como el autor, la versión y los derechos del autor.

La clase **GenericServlet** es una implementación genérica de esta interfaz y no suele usarse. La clase **HttpServlet** es una extensión de **GenericServlet** y está diseñada específicamente para manejar el protocolo HTTP —**HttpServlet** es la que se usará la mayoría de veces.

El atributo más conveniente de la API de servlets lo constituyen los objetos auxiliares que vienen con la clase **HttpServlet** para darle soporte. Si se mira al método **service( )** de la interfaz **Servlet**, se verá que tiene dos parámetros: **ServletRequest** y **ServletResponse**. Con la clase **HttpServlet** se extienden estos dos objetos para HTTP: **HttpServletRequest** y **HttpServletResponse**. He aquí un ejemplo simple que muestra el uso de **HttpServletResponse**:

```
//: c15:servlets:ReglasServlets.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ReglasServlets extends HttpServlet {
  int i = 0; // Servlet "persistencia"
  public void service(HttpServletRequest req,
  HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter salida = res.getWriter();
    salida.print("<HEAD><TITLE>");
    salida.print("Una estrategia de lado servidor");
```

```
      salida.print("</TITLE></HEAD><BODY>");
      salida.print("<h1>¡Reglas Servets! " + i++);
      salida.print("</h1></BODY>");
      salida.close();
   }
} ///:~
```

ReglasServlets es casi tan simple como puede serlo un servlet. El servlet sólo se inicializa una vez llamando a su método **init( )**, al cargar el servidor tras arrancar por primera vez el contenedor de servlets. Cuando un cliente hace una petición a una URL que resulta representar un servlet, el contenedor de servlets intercepta la petición y hace una llamada al método **service( )**, tras configurar los objetos **HttpServletRequest** y **HttpServletResponse**.

La responsabilidad principal del método **service( )** es interactuar con la petición HTTP que ha enviado el cliente, y construir una respuesta HTTP basada en los atributos contenidos dentro de la petición. **ReglasServlets** sólo manipula el objeto respuesta sin mirar a lo que el cliente puede haber enviado.

Tras configurar el tipo de contenido de la respuesta (cosa que debe hacerse siempre antes de procurar el **Writer** u **OutputStream**), el método **getWriter( )** del objeto respuesta produce un objeto **PrintWriter**, que se usa para escribir información de respuesta basada en caracteres (alternativamente, **getOutputStream( )** produce un **OutputStream**, usado para respuesta binaria, que sólo se usa en soluciones más especializadas).

El resto del programa simplemente manda HTML de vuelta al cliente (se asume que el lector entiende HTML, por lo que no se explica esa parte) como una secuencia de **Strings**. Sin embargo, nótese la inclusión del "contador de accesos" representado por la variable **i**. Éste se convierte automáticamente en un **String** en la sentencia **print( )**.

Cuando se ejecuta el programa, se verá que se mantiene el valor de **i** entre las peticiones al servidor. Ésta es una propiedad esencial de los servlets: puesto que en el contenedor sólo se carga un servlet de cada clase, y éste nunca se descarga (a menos que finalice el contenedor de servlets, lo cual es algo que normalmente sólo ocurre si se reinicia la máquina servidora), ¡cualquier campo de esa clase servidora se convierte en un objeto persistente! Esto significa que se pueden mantener sin esfuerzo valores entre peticiones servlets, mientras que con CGI había que escribir los valores al disco para preservarlos, lo que requería una cantidad de trabajo bastante elevada para que funcionara con éxito, y solía producir soluciones exclusivas para una plataforma.

Por supuesto, en ocasiones, el servidor web, y por consiguiente, el contenedor de servlets, tienen que ser reiniciados como parte del mantenimiento o por culpa de un fallo de corriente. Para evitar perder cualquier información persistente, se invoca automáticamente a los métodos **init( )** y **destroy( )** del servlet siempre que se carga o descarga el servlet, proporcionando la oportunidad de salvar los datos durante el apagado, y restaurarlos tras el nuevo arranque. El contenedor de servlets llama al método **destroy( )** al terminarse a sí mismo, por lo que siempre se logra una oportunidad de salvar la información valiosa, en la medida en que la máquina servidora esté configurada de forma inteligente.

Hay otro aspecto del uso de **HttpServlet**. Esta clase proporciona métodos **doGet( )** y **doPost( )**, que diferencian entre un envío CGI "GET" del cliente, y un CGI "POST". GET y POST simplemente varían en los detalles de la forma en que envían los datos, que es algo que preferimos ignorar. Sin embargo, la mayoría de información publicada que hemos visto parece ser favorable a la creación de métodos **doGet( )** y **doPost( )** separados en vez de un método **service( )** genérico, que maneje los dos casos. Este favoritismo parece bastante común, pero nunca lo he visto explicado de forma que nos haga creer que se deba a algo más que a la inercia de los programadores de CGI que están habituados a prestar atención a si se está usando GET o POST. Por tanto, y por mantener el espíritu de "hacer todo siempre de la forma más simple que funcione"[4], simplemente usaremos el método **service( )** en estos ejemplos, y que se encargue de los GET *frente a* POST. Sin embargo, hay que mantener presente que podríamos dejarnos algo, por lo que de hecho sí que podría haber una buena razón para usar en su lugar **doGet( )** o **doPost( )**.

Siempre que se envía un formulario a un servidor, el **HttpServletRequest** viene precargado con todos los datos del formulario, almacenados como pares clave-valor. Si se conocen los nombres de los campos, pueden usarse directamente con el método **getParameter( )** para buscar los valores. También se puede lograr un **Enumeration** (la forma antigua del **Iterator**) para los nombres de los campos, como se muestra en el ejemplo siguiente. Este ejemplo también demuestra cómo se puede usar un único servlet para producir la página que contiene el formulario y para responder a la pagina (más adelante se verá una solución mejor, con JSP). Si la **Enumeration** está vacía, no hay campos; esto significa que no se envió ningún formulario. En este caso, se produce el formulario y el botón de enviar reinvocará al mismo servlet. Sin embargo, si los campos existen, se muestran.

```
/: c15:servlets:EcoFormulario.java
// Vuelca los pares nombre-valor de cualquier
// formulario HTML
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EcoFormulario extends HttpServlet {
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter salida = res.getWriter();
    Enumeration campos = req.getParameterNames();
    if(!campos.hasMoreElements()) {
      // No se envía formulario -crear uno:
      salida.print("<html>");
      salida.print("<form method=\"POST\"" +
        " action=\"EcoFormulario\">");
      for(int i = 0; i < 10; i++)
```

[4] Uno de los eslóganes principales de la Programación Extrema (XP). Ver *http:www.xprogramming.com.*

```
          salida.print("<b>Campo" + i + "</b> " +
            "<input type=\"text\""+
            " size=\"20\" name=\"Campo" + i +
            "\" value=\"Valor" + i + "\"><br>");
        salida.print("<INPUT TYPE=SUBMIT name=someter"+
        " Value=\"Someter\"></form></html>");
      } else {
        salida.print("<h1>Tu formulario contenia:</h1>");
        while(campos.hasMoreElements()) {
          String campo= (String)campos.nextElement();
          String valor= req.getParameter(campo);
          salida.print(campo + " = " + valor+ "<br>");
        }
      }
      salida.close();
    }
} ///:~
```

Una pega que se verá aquí es que Java no parece haber sido diseñado con el procesamiento de cadenas de caracteres en mente —el formateo de la página de retorno no supone más que quebraderos de cabeza debido a los saltos de línea, las marcas de escape y los signos "+" necesarios para construir objetos **String**. Con una página HTML extensa no sería razonable codificarla directamente en Java. Una solución es mantener la página como un archivo de texto separado, y abrirla y pasársela al servidor web. Si se tiene que llevar a cabo cualquier tipo de sustitución de los contenidos de la página, la solución no es mucho mejor debido al procesamiento tan pobre de las cadenas de texto en Java. En estos casos, probablemente se hará mejor usando una solución más apropiada (nuestra elección sería Phyton; hay una versión que se fija en Java llamada JPython) para generar la página de respuesta.

Servlets y multihilo

El contenedor de servlets tiene un conjunto de hilos que irá despachando para gestionar las peticiones de los clientes. Es bastante probable que dos clientes que lleguen al mismo tiempo puedan ser procesados por **service( )** a la vez. Por consiguiente, el método **service( )** debe estar escrito de forma segura para hilos. Cualquier acceso a recursos comunes (archivos, bases de datos) necesitará estar protegido haciendo uso de la palabra clave **synchronized**.

El siguiente ejemplo simple pone una cláusula **synchronized** en torno al método **sleep( )** del hilo. Éste bloqueará al resto de los hilos hasta que se haya agotado el tiempo asignado (5 segundos). Al probar esto, deberíamos arrancar varias instancias del navegador y acceder a este servlet tan rápido como se pueda en cada una —se verá que cada una tiene que esperar hasta que le llega su turno.

```
//: c15:servlets:HiloServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class HiloServlet extends HttpServlet {
  int i;
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter salida = res.getWriter();
    synchronized(this) {
      try {
        Thread.currentThread().sleep(5000);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
    salida.print("<h1>Finalizado " + i++ + "</h1>");
    salida.close();
  }
} ///:~
```

También es posible sincronizar todo el servlet poniendo la palabra clave **synchronized** delante del método **service( )**. De hecho, la única razón de usar la cláusula **synchronized** en su lugar es por si la sección crítica está en un cauce de ejecución que podría no ejecutarse. En ese caso, se podría evitar también la sobrecarga de tener que sincronizar cada vez utilizando una cláusula **synchronized**. De otra forma, todos los hilos tendrían que esperar de todas formas, por lo que también se podría **sincronizar** todo el método.

Gestionar sesiones con servlets

HTTP es un protocolo "sin sesión", por lo que no se puede decir desde un acceso al servidor a otro si se trata de la misma persona que está accediendo repetidamente al sitio, o si se trata de una persona completamente diferente. Se ha invertido mucho esfuerzo en mecanismos que permitirán a los desarrolladores web llevar a cabo un seguimiento de las sesiones. Las compañías no podrían hacer comercio electrónico sin mantener un seguimiento de un cliente y por ejemplo, de los elementos que éste ha introducido en su carro de la compra.

Hay bastantes métodos para llevar a cabo el seguimiento de sesiones, pero el más común es con "*cookies*" persistentes, que son una parte integral de los estándares Internet. El Grupo de Trabajo HTTP de la Internet Engineering Task Force ha descrito las *cookies* en el estándar oficial en RFC 2109 (*ds.internic.net/rfc/rfc 2109.txt* o compruebe *www.cookiecentral.com*).

Una *cookie* no es más que una pequeña pieza de información enviada por un servidor web a un navegador. El navegador almacena la cookie en el disco local y cuando se hace otra llamada al URL con la que está asociada la *cookie*, éste se envía junto con la llamada, proporcionando así que la información deseada vuelva a ese servidor (generalmente, proporcionando alguna manera de que el servidor pueda determinar que es uno el que llama). Los clientes, sin embargo, pueden desactivar la habilidad del navegador para aceptar *cookies*. Si el sitio debe llevar un seguimiento de un cliente que ha desactivado las *cookies*, hay que incorporar a mano otro método de seguimiento de sesiones

(reescritura de URL o campos de formulario ocultos), puesto que las capacidades de seguimiento de sesiones construidas en el API servlet están diseñadas para *cookies*.

La clase **Cookie**

El API servlet (en versiones 2.0 y superior) proporciona la clase **Cookie**. Esta clase incorpora todos los detalles de cabecera HTTP y permite el establecimiento de varios atributos de *cookie*. Utilizar la *cookie* es simplemente un problema de añadirla al objeto respuesta. El constructor toma un nombre de *cookie* como primer parámetro y un valor como segundo. Las cookies se añaden al objeto respuesta antes de enviar ningún contenido.

```
Cookie oreo = new Cookie("TIJava", "2000");
res.addCookie(cookie);
```

Las *cookies* suelen recubrirse invocando al método **getCookies( )** del objeto **HttpServletRequest**, que devuelve un array de objetos *cookie*.

```
Cookie[] cookies = req.getCookies();
```

Después se puede llamar a **getValue( )** para cada cookie, para producir un **String** que contenga los contenidos de la *cookie*. En el ejemplo de arriba, **getValue("TIJava")** producirá un **String** de valor "2000".

La clase **Session**

Una sesión es una o más solicitudes de páginas por parte de un cliente a un sitio web durante un periodo definido de tiempo. Si uno compra, por ejemplo, ultramarinos en línea, se desea que una sesión dure todo el periodo de tiempo desde que se añade al primer elemento a "Mi carrito de la compra" hasta el momento en el que se compruebe todo. Cada elemento que se añada al carrito de la compra vendrá a producir una nueva conexión HTTP, que no tiene conocimiento de conexiones previas o de los elementos del carrito de la compra. Para compensar esta falta de información, los mecanismos suministrados por la especificación de *cookies* permiten al servlet llevar a cabo seguimiento de sesiones.

Un objeto **Session** servlet vive en la parte servidora del canal de comunicación; su meta es capturar datos útiles sobre este cliente a medida que el cliente recorre e interactúa con el sitio web. Esta información puede ser pertinente para la sesión actual, como los elementos del carro de la compra, o pueden ser datos como la información de autentificación introducida cuando el cliente entró por primera vez en el sitio web, y que no debería ser reintroducida durante un conjunto de transacciones particular.

La clase **Session** del API servlet usa la clase **Cookie** para hacer su trabajo. Sin embargo, todo el objeto **Session** necesita algún tipo de identificador único almacenado en el cliente y que se pasa al servidor. Los sitios web también pueden usar otros tipos de seguimiento de sesión, pero estos mecanismos serán más difíciles de implementar al no estar encapsulados en el API servlet (es decir, hay que escribirlos a mano para poder enfrentarse a la situación de deshabilitación de las *cookies* por parte del cliente).

He aquí un ejemplo que implementa seguimiento de sesión con el API servlet:

```
//: c15:servlets:SeguirSesion.java
// Usando la clase HttpSession.
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SeguirSesion extends HttpServlet {
  public void service(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException, IOException {
    // Retirar el objeto Session antes de enviar
    // ninguna salida al cliente.
    HttpSession sesion = req.getSession();
    res.setContentType("text/html");
    PrintWriter salida = res.getWriter();
    salida.println("<HEAD><TITLE> SeguirSesion ");
    salida.println(" </TITLE></HEAD><BODY>");
    salida.println("<h1> SeguirSesion </h1>");
    // Un contador de accesos simple para esta sesion.
    Integer ival = (Integer)
      sesion.getAttribute("sesspeek.cntr");
    if(ival==null)
      ival = new Integer(1);
    else
      ival = new Integer(ival.intValue() + 1);
    sesion.setAttribute("sesspeek.cntr", ival);
    salida.println("Has accedido a esta página <b>"
      + ival + "</b> veces.<p>");
    salida.println("<h2>");
    salida.println("Grabados datos de la sesion </h2>");
    // Iterar por todos los datos de la sesión:
    Enumeration nombresSes =
      sesion.getAttributeNames();
    while(nombresSes.hasMoreElements()) {
      String nombre =
        nombresSes.nextElement().toString();
      Object valor = sesion.getAttribute(nombre);
      salida.println(name + " = " + value + "<br>");
    }
    salida.println("<h3> Estadisticas de la sesion </h3>");
    salida.println("ID Sesion : "
      + sesion.getId() + "<br>");
```

```
    salida.println("Nueva Sesion: " + sesion.isNew()
      + "<br>");
    salida.println("Hora de Creacion: "
      + sesion.getCreationTime());
    salida.println("<I>(" +
      new Date(sesion.getCreationTime())
      + ")</I><br>");
    salida.println("Hora del ultimo acceso: " +
      sesion.getLastAccessedTime());
    salida.println("<I>(" +
      new Date(sesion.getLastAccessedTime())
      + ")</I><br>");
    salida.println("Intervalo de Inactividad de la sesion: "
      + sesion.getMaxInactiveInterval());
    salida.println("ID de sesion en peticion: "
      + req.getRequestedSessionId() + "<br>");
    salida.println("ID de sesion desde Cookie: "
      + req.isRequestedSessionIdFromCookie()
      + "<br>");
    salida.println("Es el ID de la session del URL: "
      + req.isRequestedSessionIdFromURL()
      + "<br>");
    salida.println("Es ID de session valido: "
      + req.isRequestedSessionIdValid()
      + "<br>");
    salida.println("</BODY>");
    salida.close();
  }
  public String getServletInfo() {
    return "Un servlet de seguimiento de sesion";
  }
} ///:~
```

Dentro del método **service( )**, se invoca a **getSession( )** para el objeto petición, que devuelve el objeto **Session** asociado con esta petición. El objeto **Session** no viaja a través de la red, sino que en vez de ello, vive en el servidor asociado con un cliente y sus peticiones.

El método **getSession( )** viene en dos versiones: la de sin parámetros, usada aquí, y **getSession(boolean)**. Usar **getSession(true)** equivale a **getSession( )**. La única razón del **boolean** es para establecer si se desea crear el objeto sesión si no es encontrado. La llamada más habitual es **getSession(true)**, razón de la existencia de **getSession( )**.

El objeto **Session**, si no es nuevo, nos dará detalles sobre el cliente provenientes de sus visitas anteriores. Si el objeto **Session** es nuevo, el programa comenzará a recopilar información sobre las actividades del cliente en esta visita. La captura de esta información del cliente se hace mediante los métodos **setAttribute( )** y **getAttribute( )** del objeto de sesión.

```
java.lang.Object getAtribute(java.lang.String)
void setAttribute(java.lang.String nombre,
                  java.lang.Object valor)
```

El objeto **Session** utiliza un emparejamiento nombre-valor simple para cargar información. El nombre es un **String**, y el valor puede ser cualquier objeto derivado de **java.lang.Object**. **SeguirSesion** mantiene un seguimiento de las veces que ha vuelto el cliente durante esta sesión. Esto se hace con un objeto **Integer** denominado **sesspeek.cntr**. Si no se encuentra el nombre se crea un **Integer** de valor uno, si no, se crea un **Integer** con el valor incrementado respecto del **Integer** anteriormente guardado. Si se usa la misma clave en una llamada a **setAttribute( )**, el objeto nuevo sobreescribe el viejo. El contador incrementado se usa para mostrar el número de veces que ha vistado el cliente durante esta sesión.

El método **getAttributeNames( )** está relacionado con **getAttribute( )** y **setAttribute( )**; devuelve una enumeración de los nombres de objetos vinculados al objeto **Session**. Un bucle **while** en **SeguirSesion** muestra este método en acción.

Uno podría preguntarse durante cuánto tiempo puede permanecer inactivo un objeto **Session**. La respuesta depende del contenedor de servlets que se esté usando; generalmente vienen por defecto a 30 minutos (1.800 segundos), que es lo que debería verse desde la llamada a **SeguirSesion** hasta **getMaxInactiveInterval( )**. Las pruebas parecen producir resultados variados entre contenedores de servlets. En ocasiones, el objeto **Session** puede permanecer inactivo durante toda la noche, pero nunca hemos visto ningún caso en el que el objeto **Session** desaparezca en un tiempo menor al especificado por el intervalo de inactividad. Esto se puede probar estableciendo el valor de este intervalo con **setMaxInactiveInterval( )** a 5 segundos y ver si el objeto **Session** se cuelga o es eliminado en el tiempo apropiado. Éste puede constituir un atributo a investigar al seleccionar un contenedor de servlets.

Ejecutar los ejemplos de servlets

Si el lector no está trabajando con un servidor de aplicaciones que maneje automáticamente las tecnologías servlet y JSP de Sun, puede descargar la implementación Tomcat de los servlets y JSPs de Java, que es una implementación gratuita, de código fuente abierto, y que es la implementación de referencia oficial de Sun. Puede encontrarse en *jakarta.apache.org*.

Siga las instrucciones de instalación de la implementación Tomcat, después edite el archivo **server.xml** para que apunte a la localización de su árbol de directorios en el que se ubicarán los servlets. Una vez que se arranque el programa Tomcat, se pueden probar los programas de servlets.

Ésta sólo ha sido una somera introducción a los servlets; hay libros enteros sobre esta materia. Sin embargo, esta introducción debería proporcionarse las suficientes ideas como para que se inicie. Además, muchas ideas de la siguiente sección son retrocompatibles con los servlets.

Java Server Pages

Las *Java Server Pages* (JSP) son una extensión del estándar Java definido sobre las Extensiones de servlets. La meta de las JSP es la creación y gestión simplificada de páginas web dinámicas.

La implementación de referencia Tomcat anteriormente mencionada y disponible gratuitamente en *jakarta.apache.org* soporta JSP automáticamente.

Las JSP permite combinar el HTML de una página web con fragmentos de código Java en el mismo documento. El código Java está rodeado de etiquetas especiales que indican al contenedor de JSP que debería usar el código para generar un servlet o parte de uno. El beneficio de las JSP es que se puede mantener un documento único que representa tanto la página como el código Java que habilita. La pega es que el mantenedor de la página JSP debe dominar tanto HTML como Java (sin embargo, los entornos constructores de IGU para JSP deberían aparecer en breve).

La primera vez que el contenedor de JSP carga un JSP (que suele estar asociado con, o ser parte de, un servidor web) se genera, compila y carga automáticamente en el contenedor de servlets el código servlet necesario para cumplimentar las etiquetas JSP. Las porciones estáticas de la página HTML se producen enviando objetos **String** estáticos a **write( )**. Las porciones dinámicas se incluyen directamente en el servlet.

A partir de ese momento, y mientras el código JSP de la página no se modifique, se comporta como si fuera una página HTML estática con servlets asociados (sin embargo, el servlet genera todo el código HTML). Si se modifica el código fuente de la JSP, ésta se recompila y recarga automáticamente la siguiente vez que se solicite esa página. Por supuesto, debido a todo este dinamismo, se apreciará una respuesta lenta en el acceso por primera vez a una JSP. Sin embargo, dado que una JSP suele usarse mucho más a menudo que ser cambiada, normalmente uno no se verá afectado por este retraso.

La estructura de una página JSP está a caballo entre la de un servlet y la de una página HTML. Las etiquetas JSP empiezan y acaban con “<” y “>”, como las etiquetas HTML, pero las etiquetas también incluyen símbolos de porcentaje, de forma que todas las etiquetas JSP se delimitan por:

```
<% código JSP aquí%>
```

El signo de porcentaje precedente puede ir seguido de otros caracteres que determinen el tipo específico de código JSP de la etiqueta.

He aquí un ejemplo extremadamente simple que usa una llamada a la biblioteca estándar Java para lograr la hora actual en milisegundos, que es después dividida por mil para producir la hora en segundos. Dado que se usa una *expresión JSP* (la <%=), el resultado del cálculo se fuerza a un **String** y se coloca en la página web generada:

```
//:! C15:jsp:MostrarSegundos.jsp
<html><body>
<H1>El tiempo en segundos es:
<%= System.currentTimeMillis()/1000 %></H1>
```

```
</body></html>
///:~
```

En los ejemplos de JSP de este libro no se incluyen la primera y última líneas en el archivo de código extraído y ubicado en el árbol de códigos fuente del libro.

Cuando el cliente crea una petición de la página JSP hay que haber configurado el servidor web para confiar en la petición del contenedor de JSP que posteriormente invoca a la página. Como se mencionó arriba, la primera vez que se invoca la página, el contenedor de JSP genera y compila los componentes especificados por la página como uno o más servlets. En el ejemplo de arriba, el servlet contendrá código para configurar el objeto **HttpServletResponse**, producir un objeto **PrintWriter** (que siempre se denomina **out**), y después convertir el cómputo de tiempo en un **String** que es enviado a **out**. Como puede verse, todo esto se logra con una sentencia muy sucinta, pero el programador HTML/diseñador web medio no tendrá las aptitudes necesarias para escribir semejante código.

Objetos implícitos

Los servlets incluyen clases que proporcionan utilidades convenientes, como **HttpServletRequest**, **HttpServletResponse, Session**, etc. Los objetos de estas clases están construidos en la especificación JSP y automáticamente disponibles para ser usados en un JSP, sin tener que escribir ninguna línea extra de código. Los objetos implícitos de un JSP se detallan en la tabla siguiente:

Variable implícita	De tipo (javax.servlet)	Descripción	Ámbito
request	Subtipo de protocolo dependiente de **HttpServletRequest**	La petición que dispara la invocación del servicio.	petición
response	Subtipo de protocolo dependiente de **HttpServletResponse**	La respuesta a la petición.	página
pageContext	**jsp.PageContext**	El contexto de la página encapsula facetas dependientes de la implementación y proporciona métodos de conveniencia y acceso de espacio de nombres a este JSP.	página
session	Subtipo de protocolo dependiente de **http.HttpSession**	El objeto sesión creado para el cliente que hace la petición. Ver el objeto servlet Session.	sesión

Variable implícita	De tipo (javax.servlet)	Descripción	Ámbito
application	**ServletContext**	El contexto de servlet obtenido del objeto de configuración del servlet (por ejemplo, **getServletConfig( ), getContext( )**).	aplicación
out	**jsp.JspWriter**	El objeto que escribe en el flujo de salida.	página
config	**ServletConfig**	El **ServletConfig** de este JSP.	página
page	**java.lang.Object**	La instancia de la clase de implementación de esta página que procese la petición actual.	página

El ámbito de cada objeto puede variar significativamente. Por ejemplo, el objeto **session** tiene un ámbito que excede al de una página, pues puede abarcar varias peticiones de clientes y páginas. El objeto **application** puede proporcionar servicios a un grupo de páginas JSP que representan juntas una aplicación web.

Directivas JSP

Las directivas son mensajes al contenedor de JSP y se delimintan por la "@":

```
<%@ directiva {atr="valor"}* %>
```

Las directivas no envían nada al flujo **out**, pero son importantes al configurar los atributos y dependencias de una página JSP con el contenedor de JSP. Por ejemplo, la línea:

```
<% page language="java" %>
```

dice que el lenguaje de escritura de guiones que se está usando en la página JSP es Java. De hecho, la especificación de Java *sólo* describe las semánticas de guiones para atributos de lenguaje iguales a "Java". La intención de esta directiva es aportar flexibilidad a la tecnología JSP. En el futuro, si hubiera que elegir otro lenguaje, como Python (un buen lenguaje de escritura de guiones), entonces este lenguaje debería soportar el Entorno de Tiempo de Ejecución de Java, exponiendo el modelo de objetos de la tecnología Java al entorno de escritura de guiones, especialmente las variables implícitas definidas arriba, las propiedades de los JavaBeans y los métodos públicos.

La directiva más importante es la directiva de página. Define un número de atributos dependientes de la página y comunica estos atributos al contenedor de JSP. Entre estos atributos se incluye: **language**, **extends**, **import**, **session**, **buffer**, **autoFlush**, **isThreadSafe**, **info** y **errorPage**. Por ejemplo:

```
<%@ page session="true" import="java.util.*" %>
```

La primera línea indica que la página requiere participación en una sesión HTTP. Dado que no hemos establecido directiva de lenguaje, el contenedor de JSP usa por defecto Java y la variable de lenguaje de escritura denominada **session** es de tipo **javax.servlet.http.HttpSession**. Si la directiva hubiera sido falsa, la variable implícita **session** no habría estado disponible. Si no se especifica la variable **session**, se pone a "true" por defecto.

El atributo **import** describe los tipos disponibles al entorno de escritura de guiones. Este atributo se usa igual que en el lenguaje de programación Java, por ejemplo, una lista separada por comas de expresiones **import** normales. Esta lista es importada por la implementación de la página JSP traducida y está disponible para el entorno de escritura de guiones. De nuevo, esto sólo está definido verdaderamente cuando el valor de la directiva de lenguaje es "java".

Elementos de escritura de guiones JSP

Una vez que se han usado las directivas para establecer el entorno de escritura de guiones se puede usar los elementos del lenguaje de escritura de guiones. JSP 1.1 tiene tres elementos de lenguaje de escritura de guiones —*declaraciones*, *scriptlets* y *expresiones*. Una declaración declarará elementos, un scriptlet es un fragmento de sentencia y una expresión es una expresión completa del lenguaje. En JSP cada elemento de escritura de guiones empieza por "<%". La sintaxis de cada una es:

```
<%! declaracion %>
<% scriptlet %>
<%= expresión %>
```

El espacio en blanco tras "<%!", "<%", "<%=" y antes de "%>" es opcional.

Todas estas etiquetas se basan en XML; se podría incluso decir que una página JSP puede corresponderse con un documento XML. La sintaxis equivalente en XML para los elementos de escritura de guiones de arriba sería:

```
<jsp:declaracion> declaracion  </jsp:declaracion>
<jsp:scriptlet>  scriptlet   </jsp:scriptlet>
<jsp:expresion>  expresion </jsp:expresion>
```

Además, hay dos tipos de comentarios:

```
<%--comentario jsp --%>
<!--comentario html  -->
```

La primera forma permite añadir comentarios a las páginas fuente JSP que no aparecerán de ninguna forma en el HTML que se envía al cliente. Por supuesto, la segunda forma de comentario no es específica de los JSP —es simplemente un comentario HTML ordinario. Lo interesante es que se puede insertar código JSP dentro de un comentario HTML, y el comentario se producirá en la página resultante, incluyendo el resultado del código JSP.

Las declaraciones se usan para declarar variables y métodos en el lenguaje de escritura de guiones (actualmente sólo Java) usado en una página JSP. La declaración debe ser una sentencia Java com-

pleta y no puede producir ninguna salida en el flujo **salida**. En el ejemplo **Hola.jsp** de debajo, las declaraciones de las variables **cargaHora**, **cargaFecha** y **conteoAccesos** son sentencias Java completas que declaran e inicializan nuevas variables.

```
//:! C15:jsp:Hola.jsp
<%-- Este comentario JSP no aparecera en el HTML generado --%>
<%-- Esto es una directiva JSP: --%>
<%@ page import="java.util.*" %>
<%-- Estas son declaraciones: --%>
<%!
     long cargaHora= System.currentTimeMillis();
     Date cargaFecha = new Date();
     int conteoAccesos = 0;
%>
<html><body>
<%-- Las siguientes lineas son el resultado de una
expression JSP insertada en el html generado;
el '=' indica una expresion JSP --%>
<H1>Esta pagina fue cargada el <%= cargaFecha %> </H1>
<H1>¡Hola, Mundo! Hoy es <%= new Date() %></H1>
<H2>He aqui un objeto: <%= new Object() %></H2>
<H2>Esta pagina ha estado activa
<%= (System.currentTimeMillis()-cargaHora)/1000 %>
segundos</H2>
<H3>Esta pagina ha sido accedida <%= ++conteoAccesos %>
veces desde <%= cargaFecha %></H3>
<%-- Un "scriptlet" que escribe a la consola
servidora y a la pagina cliente.
Notese que se require el ';': --%>
<%
    System.out.println("Adios");
    out.println("Cheerio");
%>
</body></html>
///:~
```

Cuando se ejecute este programa se verá que las variables **cargaHora**, **cargaFecha** y **conteoAccesos** guardan sus valores entre accesos a la página, por lo que son claramente campos y no variables locales.

Al final del ejemplo hay un scriptlet que escribe "Adios" a la consola servidora web y "Cheerio" al objeto **JspWriter** implícito **out**. Los scriptlets pueden contener cualquier fragmento de código que sean sentencias Java válidas. Los scriptlets se ejecutan bajo demanda en tiempo de procesamiento. Cuando todos los fragmentos de scritplet en un JSP dado se combinan en el orden en que aparecen en la página JSP, deberían conformar una sentencia válida según la definición del lenguaje de programación Java. Si producen o no alguna salida al flujo **out** depende del código del scriptlet. Uno de-

bería ser consciente de que los scriptlets pueden producir efectos laterales al modificar objetos que son visibles para ellos.

Las expresiones JSP pueden entremezclarse con el HTML en la sección central de **Hola.jsp**. Las expresiones deben ser sentencias Java completas, que son evaluadas, convertidas a un **String** y enviadas a **out**. Si el resultado no puede convertirse en un **String**, se lanza una **ClassCastException**.

Extraer campos y valores

El ejemplo siguiente es similar a uno que se mostró anteriormente en la seccion de servlets. La primera vez que se acceda a la página, se detecta que no se tienen campos y se devuelve una página que contiene un formulario, usando el mismo código que en el ejemplo de los servlets, pero en formato JSP. Cuando se envía el formulario con los campos rellenos al mismo URL de JSP, éste detecta los campos y los muestra. Ésta es una técnica brillante pues permite tener tanto la página que contiene el formulario para que el usuario la rellene como el código de respuesta para esa página en un único archivo, facilitando así la creación y mantenimiento.

```
//:! c15:jsp:MostarDatosFormulario.jsp
<%-- Tomando los datos de un formulario HTML. --%>
<%-- Este JSP tambien genera el formulario. --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>MostarDatosFormulario</H1><H3>
<%
  Enumeration campos = request.getParameterNames();
  if(!campos.hasMoreElements()) { // No hay campos %>
    <form method="POST"
    action="MostarDatosFormulario.jsp">
<%  for(int i = 0; i < 10; i++) {  %>
      Campo<%=i%>: <input type="text" size="20"
      name="Campo<%=i%>" value="Valor<%=i%>"><br>
<%  } %>
    <INPUT TYPE=submit name=someter
    value="Someter"></form>
<%} else {
    while(campos.hasMoreElements()) {
      String campo = (String)campos.nextElement();
      String valor = request.getParameter(campo);
%>
      <li><%= campo %> = <%= valor %></li>
<%  }
  } %>
</H3></body></html>
///:~
```

El aspecto más interesante de este ejemplo es que demuestra cómo puede entremezclarse el código scriptlet con el código HTML incluso hasta el punto de generar HTML dentro de un bucle **for** de Java. Esto es especialmente conveniente para construir cualquier tipo de formulario en el que, de lo contrario, se requeriría código HTML repetitivo.

Atributos JSP de página y su ámbito

Merodeando por la documentación HTML de servlets y JSP, se pueden encontrar facetas que dan información sobre el servlet o el JSP actualmente en ejecución. El ejemplo siguiente muestra uno de estos fragmentos de datos:

```
//:! c15:jsp:ContextoPagina.jsp
<%--Viendo los atributos de ContextoPagina--%>
<%-- Notese que se puede incluir cualquier cantidad de codigo
dentro de las etiquetas de scriptlet --%>
<%@ page import="java.util.*" %>
<html><body>
NombreServlet: <%= config.getServletName() %><br>
El contenedor de servlets soporta la version:
<% out.print(application.getMajorVersion() + "."
+ application.getMinorVersion()); %><br>
<%
  session.setAttribute("Mi perro", "Ralph");
  for(int ambito = 1; ambito <= 4; ambito++) {  %>
    <H3>Ambito: <%= ambito %> </H3>
<%  Enumeration e =
      pageContext.getAttributeNamesInScope(ambito);
    while(e.hasMoreElements()) {
      out.println("\t<li>" +
        e.nextElement() + "</li>");
    }
  }
%>
</body></html>
///:~
```

Este ejemplo también muestra el uso tanto del HTML embebido como de la escritura en **out** para sacar la página HTML resultante.

El primer fragmento de información que se produce es el nombre del servlet, que probablemente será simplemente "JSP", pero depende de la implementación. También se puede descubrir la versión actual del contenedor de servlets usando el objeto aplicación. Finalmente, tras establecer un atributo de sesión, se muestran los "nombres de atributo" de un ámbito en particular. Los ámbitos no se usan mucho en la mayoría de programas JSP; simplemente se muestran aquí para añadir interés al ejemplo. Hay cuatro ámbitos de atributos, que son: el ***ámbito de página*** (ámbito 1), el ***ám-***

bito de petición (ámbito 2), el *ámbito de sesión* (ámbito 3) —aquí, el único elemento disponible en ámbito de sesión es "Mi perro", añadido justo antes del bucle **for**, y el *ámbito de aplicación* (ámbito 4), basado en el objeto **ServletContext**. Sólo hay un **ServletContext** por "aplicación web" por cada Máquina Virtual Java. (Una "aplicación web" es una colección de servlets y contenido instalados bajo un subconjunto del espacio de nombres URL del servidor, como /catalog. Esto se suele establecer utilizando un archivo de configuración.) En el ámbito de aplicación se verán objetos que representan rutas para el directorio de trabajo y el directorio temporal.

Manipular sesiones en JSP

Las sesiones se presentaron en la sección anterior de los servlets, y también están disponibles dentro de los JSP. El ejemplo siguiente ejercita el objeto **session** y permite manipular la cantidad de tiempo antes de que la sesión se vuelva no válida.

```
//:! c15:ObjetoSesion.jsp
<%--Recuperando y estableciendo valores de objetos session --%>
<html><body>
<H1>IDSesion : <%= session.getId() %></H1>
<H3><li>Esta sesion se creo el
<%= session.getCreationTime() %></li></H1>
<H3><li>Intervalo Maximo de Inactividad anterior =
  <%= session.getMaxInactiveInterval() %></li>
<% session.setMaxInactiveInterval(5); %>
<li>Nuevo intervalo maximo de inactividad=
  <%= session.getMaxInactiveInterval() %></li>
</H3>
<H2>Si el objeto sesion "Mi perro" sigue vivo,
este valor sera distinto de null: <H2>
<H3><li>Valor de sesion para "Mi perro" =
<%= session.getAttribute("Mi perro") %></li></H3>
<%-- Ahora añadir el objeto sesion "Mi perro" --%>
<% session.setAttribute("Mi perro",
                        new String("Ralph")); %>
<H1>El nombre de mi perro es
<%= session.getAttribute("Mi perro") %></H1>
<%-- Ver si "Mi perro" pasa a otra forma --%>
<FORM TYPE=POST ACTION=ObjetoSesion2.jsp>
<INPUT TYPE=submit name=someter
Value="Invalidar"></FORM>
<FORM TYPE=POST ACTION=ObjetoSesion3.jsp>
<INPUT TYPE=submit name=Someter
Value="Mantener"></FORM>
</body></html>
///:~
```

El objeto **session** se proporciona por defecto, por lo que está disponible sin necesidad de codificación extra. Las llamadas a **getId( )**, **getCreationTime( )** y **getMaxInactiveInterval( )** se usan para mostrar información sobre este objeto **session**.

Cuando se trae por primera vez esta sesión se verá un **MaxInactiveInterval** de, por ejemplo, 1800 segundos (30 minutos). Esto dependerá de la forma en que esté configurado el contenedor de JSP/servlets. El **MaxInactiveInterval** se acorta a 5 segundos para que las cosas parezcan interesantes. Si se refresca la página antes de que expire el intervalo de 5 segundos, se verá:

```
Valor de sesion para "Mi perro" = Ralph
```

Pero si se espera más que eso, "Ralph" se convertirá en **null**.

Para ver cómo se puede traer la información de sesión a otras páginas, y para ver también el efecto de invalidar un objeto de sesión *frente a* simplemente dejar que expire, se crean otros dos JSP. El primero (al que se llega presionando el botón "invalidar" de **ObjetoSesion.jsp**) lee la información de sesión y después invalida esa sesión explícitamente:

```
//:! c15:jsp:ObjetoSesion2.jsp
<%--El objeto sesion se arrastra --%>
<html><body>
<H1>ID Sesion: <%= session.getId() %></H1>
<H1>Valor de sesion para "Mi perro"
<%= session.getValue("Mi perro") %></H1>
<% session.invalidate(); %>
</body></html>
///:~
```

Para experimentar con esto, refresque **ObjetoSesion.jsp**, y después pulse inmediatamente en el objeto "invalidar" para ir a **ObjetoSesion2.jsp**. En este momento se verá "Ralph", y después (antes de que haya expirado el intervalo de 5 segundos), refresque **ObjetoSesion2.jsp** para ver que la sesión ha sido invalidada a la fuerza y que "Ralph" ha desaparecido.

Si se vuelve a **ObjetoSesion.jsp**, refresque la página, de forma que se tenga un nuevo intervalo de 5 segundos, desués presione el botón "Mantener", que le llevará a la siguiente página, **ObjetoSesion3.jsp**, que NO invalida la sesión:

```
//:! c15:jsp:ObjetoSesion3.jsp
<%--El objeto sesion se arrastra --%>
<html><body>
<H1>ID Sesion: <%= session.getId() %></H1>
<H1>Valor de sesion para "Mi perro"
<%= session.getValue("Mi perro") %></H1>
<FORM TYPE=POST ACTION=ObjetoSesion.jsp>
<INPUT TYPE=submit name=someter Value="volver">
</FORM>
</body></html>
```

```
///:~
```

Dado que esta página no invalida la sesión, "Ralph" merodeará por ahí mientras se siga refrescando la página, antes de que expire el intervalo de 5 segundos. Esto es semejante a una mascota "Tomagotchi" —en la medida en que se juega con "Ralph", sigue vivo, si no, expira.

Crear y modificar cookies

Las cookies se presentaron en la sección anterior relativa a servlets. De nuevo, la brevedad de los JSP hace que jugar con las cookies aquí sea mucho más sencillo que con el uso de servlets. El ejemplo siguiente muestra esto cogiendo las cookies que vienen con la petición, leyendo y modificando sus edades máximas (fechas de expiración) y adjuntando una Cookie a la respuesta saliente:

```
//:! c15:jsp:Cookies.jsp
<%--¡Este programa tiene distintos comportamientos con
 los distintos navegadores! --%>
<html><body>
<H1>IDsesion: <%= session.getId() %></H1>
<%
Cookie[] cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++) { %>
  Cookie nombre: <%= cookies[i].getName() %> <br>
  valor: <%= cookies[i].getValue() %><br>
  Vieja edad maxima en segundos:
  <%= cookies[i].getMaxAge() %><br>
  <% cookies[i].setMaxAge(5); %>
  Nueva edad maxima en segundos:
  <%= cookies[i].getMaxAge() %><br>
<% } %>
<%! int conteo = 0; int conteop = 0; %>
<% response.addCookie(new Cookie(
    "Bob" + conteo++, "Perro" + conteop++)); %>
</body></html>
///:~
```

Dado que cada navegador almacena las cookies a su manera, se pueden ver distintos comportamientos con distintos navegadores (no puede asegurarse, pero podría tratarse de algún tipo de error solucionado para cuando se lea el presente texto). También podrían experimentarse resultados distintos si se apaga el navegador y se vuelve a arrancar en vez de visitar simplemente una página distinta y volver a **Cookies.jsp**. Nótese que usar objetos sesión parece ser más robusto que el uso directo de cookies.

Tras mostrar el identificador de sesión, se muestra cada cookie del array de cookies que viene con el objeto **request**, junto con su edad máxima. Después se cambia la edad máxima y se muestra de nuevo para verificar el nuevo valor. A continuación se añade una nueva cookie a la respuesta. Sin embargo, el navegador puede parecer ignorar la edad máxima; merece la pena jugar con este pro-

grama y modificar el valor de la edad máxima para ver el comportamiento bajo distintos navegadores.

Resumen de JSP

Esta sección sólo ha sido un recorrido breve por los JSP, e incluso con lo aquí cubierto (junto con lo que se ha aprendido en el resto del libro, y el conocimiento que cada uno tenga de HTML) se puede empezar a escribir páginas web sofisticadas vía JSP. La sintaxis de JSP no pretende ser excepcionalmente profunda o complicada, por lo que si se entiende lo que se ha presentado en esta sección, uno ya puede ser productivo con JSP. Se puede encontrar más información en los libros más actuales sobre servlets o en *java.sun.com*.

Es especialmente bonito tener disponibles los JSP, incluso si la meta es producir servlets. Se descubrirá que si se tiene una pregunta sobre el comportamiento de una faceta servlet, es mucho más fácil y sencillo escribir un programa de pruebas de JSP para responder a esa cuestión, que escribir un servlet. Parte del beneficio viene de tener que escribir menos código y de ser capaz de mezclar el HTML con el código Java, pero la mayor ventaja resulta especialmente obvia cuando se ve que el contenedor JSP maneja toda la recompilación y recarga de JSP automáticamente siempre que se cambia el código fuente.

Siendo los JSP tan terroríficos, sin embargo, merece la pena ser conscientes de que la creación de JSP requiere de un nivel de talento más elevado que el necesario para simplemente programar en Java o crear páginas web. Además, depurar una página JSP que no funciona no es tan fácil como depurar un programa Java, puesto que (actualmente) los mensajes de error son más oscuros. Esto podría cambiar al mejorar los sistemas de desarrollo, pero también puede que veamos otras tecnologías construidas sobre Java y la Web que se adapten mejor a las destrezas del diseñador de sitios web.

RMI (Invocation Remote Method)

Los enfoques tradicionales a la ejecución de código en otras máquinas a través de una red siempre han sido confusos a la vez que tediosos y fuentes de error a la hora de su implementación. La mejor forma de pensar en este problema es que algún objeto resulte que resida en otra máquina, y que se pueda enviar un mensaje al objeto remoto y obtener un resultado exactamente igual que si el objeto viviera en la máquina local. Esta simplificación es exactamente lo que permite hacer el *Remote Method Invocation* (RMI) de Java. Esta sección recorre los pasos necesarios para que cada uno cree sus propios objetos RMI.

Interfaces remotos

RMI hace un uso intensivo de las interfaces. Cuando se desea crear un objeto remoto, se enmascara la implementación subyacente pasando una interfaz. Por consiguiente, cuando el cliente obtiene una referencia a un objeto remoto, lo que verdaderamente logra es una referencia a una interfaz,

que resulta estar conectada a algún fragmento de código local que habla a través de la red. Pero no hay que pensar en esto, sino simplemente en enviar mensajes vía la referencia a la interfaz.

Cuando se cree una interfaz remota, hay que seguir estas normas:

1. La interfaz remota debe ser **public** (no puede tener "acceso package" es decir, no puede ser "amigo". De otra forma, el cliente obtendría un error al intentar cargar un objeto remoto que implemente la interfaz remota.

2. La interfaz remota debe extender la interfaz **java.rmi.Remote**.

3. Cada método de la interfaz remota debe declarar **java.rmi.RemoteException** en su cláusula **throws,** además de cualquier excepción específica de la aplicación.

4. Todo objeto remoto pasado como parámetro o valor de retorno (bien directamente o bien embebido en un objeto local) debe declararse como la interfaz remota, no como la clase implementación.

He aquí una interfaz remota simple que representa un servicio de tiempo exacto:

```
//: c15:rmi:ITiempoPerfecto.java
// La interfaz remota TiempoPerfecto.
package c15.rmi;
import java.rmi.*;

interface ITiempoPerfecto extends Remote {
  long obtenerTiempoPerfecto() throws RemoteException;
} ///:~
```

Tiene el mismo aspecto que cualquier otra interfaz, excepto por extender **Remote** y porque todos sus métodos lanzan **RemoteException**. Recuérdese que una **interfaz** y todos sus métodos son automáticamente **public**.

Implementar la interfaz remota

El servidor debe contener una clase que extienda **UnicastRemoteObject** e implementar la interfaz remota. Esta clase también puede tener métodos adicionales, pero sólo los métodos de la interfaz remota están disponibles al cliente, por supuesto, dado que el cliente sólo obtendrá una referencia al interfaz, y no a la clase que lo implementa.

Hay que definir explícitamente el constructor para el objeto remoto, incluso si sólo se está definiendo un constructor por defecto que invoque al constructor de la clase base. Hay que escribirlo puesto que debe lanzar **RemoteException**.

He aquí la implementación de la interfaz remota **ITiempoPerfecto:**

```
//: c15:rmi:TiempoPerfecto.java
// La implementación del objeto
```

```
// remoto TiempoPerfecto.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class TiempoPerfecto
    extends UnicastRemoteObject
    implements ITiempoPerfecto {
  // Implementación de la interfaz:
  public long obtenerTiempoPerfecto()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Debe implementar el constructor
  // para lanzar RemoteException:
  public TiempoPerfecto() throws RemoteException {
    // super(); // Invocado automáticamente
  }
  // Registro para el servicio RMI. Lanza
  // excepciones a la consola.
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
    new RMISecurityManager());
    TiempoPerfecto tp = new TiempoPerfecto();
    Naming.bind(
      "//pepe:2005/TiempoPerfecto", tp);
    System.out.println("Preparado para dar la hora");
  }
} ///:~
```

Aquí, **main( )** maneja todos los detalles de establecimiento del servidor. Cuando se sirven objetos RMI, en algún momento del programa hay que:

1. Crear e instalar un gestor de seguridad que soporte RMI. El único disponible para RMI como parte de la distribución JAVA es **RMISecurityManager**.

2. Crear una o más instancias de un objeto remoto. Aquí, puede verse la creación del objeto **TiempoPerfecto**.

3. Registrar al menos uno de los objetos remotos con el registro de objetos remotos RMI para propósitos de reposición. Un objeto remoto puede tener métodos que produzcan referencias a otros objetos remotos. Esto permite configurarlo de forma que el cliente sólo tenga que ir al registro una vez para lograr el primer objeto remoto.

Configurar el registro

Aquí se ve una llamada al método **static Naming.bind( )**. Sin embargo, esta llamada requiere que el registro se esté ejecutando como un proceso separado en el computador. El nombre del servidor de registros es **rmiregistry**, y bajo Windows de 32 bits se dice:

```
start rmiregistry
```

para que arranque en segundo plano. En Unix, el comando es:

```
rmiregistry &
```

Como muchos programas de red, el **rmiregistry** está ubicado en la dirección IP de cualquier máquina que lo arranque, pero también debe estar escuchando por un puerto. Si se invoca al **rmiregistry** como arriba, sin argumentos, el puerto del registro por defecto será 1099. Si se desea que esté en cualquier otro puerto, se añade un argumento a la línea de comandos para especificar el puerto. Para este ejemplo, el puerto está localizado en el 2005, de forma que bajo Windows de 32 bits el **rmiregistry** debería empezarse así:

```
start rmiregistry 2005
```

o en el caso de Unix:

```
rmiregistry 2005 &
```

La información sobre el puerto también debe proporcionarse al comando **bind( )**, junto con la dirección IP de la máquina en la que está ubicado el registro. Pero esto puede ser un problema frustrante si se desea probar programas RMI de forma local de la misma forma en que se han probado otros programas de red hasta este momento en el presente capítulo. En la versión 1.1 del JDK, hay un par de problemas[5]:

1. **localhost** no funciona con RMI. Por consiguiente, para experimentar con RMI en una única máquina, hay que proporcionar el nombre de la máquina. Para averiguar el nombre de la máquina bajo Windows de 32 bits, se puede ir al panel de control y seleccionar "Red". Después, se selecciona la solapa "Identificación", y se dispondrá del nombre del computador. En nuestro caso, llamamos a mi computador "Pepe". El uso de mayúsculas y minúsculas parece ignorarse.

2. RMI no funcionará a menos que el computador tenga una conexión TCP/IP activa, incluso si todos los componentes simplemente se comunican entre sí en la máquina local. Esto significa que hay que conectarse al proveedor de servicios de Internet antes de intentar ejecutar el programa o se obtendrán algunos mensajes de excepción siniestros.

Con todo esto en mente, el comando **bind( )** se convierte en :

```
Naming.bind("//pepe:2005/TiempoPerfecto", tp);
```

---

[5] Para descubrir esta información fueron muchas las neuronas que sufrieron una muerte agónica.

Si se está usando el puerto por defecto, el 1099, no hay que especificar un puerto, por lo que podría decirse:

```
Naming.bind("//pepe/TiempoPerfecto", tp);
```

Se deberían poder hacer pruebas locales usando sólo el identificador:

```
Naming.bind("TiempoPerfecto", tp);
```

El nombre del servicio es arbitrario; resulta que en este caso es TiempoPerfecto, exactamente igual que el nombre de la clase, pero se le podría dar el nombre que se desee. Lo importante es que sea un nombre único en el registro que el cliente conozca, para buscar el objeto remoto. Si el nombre ya está en el registro, se obtiene una **AlreadyBoundException**. Para evitar esto, se puede usar siempre **rebind( )** en vez de **bind( )**, puesto que **rebind( ),** o añade una nueva entrada o reemplaza una ya existente.

Incluso aunque exista **main( )**, el objeto se ha creado y registrado, por lo que se mantiene vivo por parte del registro, esperando a que venga un cliente y lo solicite. Mientras se esté ejecutando el **rmiregistry** y no se invoque a **Naming.unbind( )** para ese nombre, el objeto estará ahí. Por esta razón, cuando se esté desarrollando código hay que apagar el **rmiregistry** y volver a arrancarlo al compilar una nueva versión del objeto remoto.

Uno no se ve forzado a arrancar **rmiregistry** como un proceso externo. Si se sabe que una aplicación es la única que va a usar el registro, se puede arrancar dentro del programa con la línea:

```
LocateRegistry.createRegistry(2005);
```

Como antes, 2005 es el número de puerto que usamos en este ejemplo. Esto equivale a ejecutar **rmiregistry** 2005 desde la línea de comandos, pero a menudo puede ser más conveniente cuando se esté desarrollando código RMI, pues elimina los pasos adicionales de arrancar y detener el registro. Una vez ejecutado este código, se puede invocar **bind( )** usando **Naming** como antes.

Crear stubs y skeletons

Si se compila y ejecuta **TiempoPerfecto.java**, no funcionará incluso aunque el **rmiregistry** se esté ejecutando correctamente. Esto se debe a que todavía no se dispone de todo el marco para RMI. Hay que crear primero los *stubs* y *skeletons* que proporcionan las operaciones de conexión de red y que permiten fingir que el objeto remoto es simplemente otro objeto local de la máquina.

Lo que ocurre tras el telón es complejo. Cualquier objeto que se pase o que sea devuelvo por un objeto remoto debe **implementar Serializable** (si se desea pasar referencias remotas en vez de objetos enteros, los parámetros objeto pueden **implementar Remote**), por lo que se puede imaginar que los *stubs y skeletons* están llevando a cabo operaciones de serialización y deserialización automáticas, al ir mandando todos los parámetros a través de la red, y al devolver el resultado. Afortunadamente, no hay por qué saber nada de esto, pero *sí* que hay que crear los *stubs* y *skeletons*. Este proceso es simple: se invoca a la herramienta **rmic** para el código compilado, y ésta crea los archivos necesarios. Por tanto, el único requisito es añadir otro paso al proceso de compilación.

Sin embargo, la herramienta **rmic** tiene un comportamiento particular para paquetes y *classpaths*. **TiempoPerfecto.java** está en el **package c15.rmi**, e incluso si se invoca a **rmic** en el mismo directorio en el que está localizada **TiempoPerfecto.class**, **rmic** no encontrará el archivo, puesto que busca el *classpath*. Por tanto, hay que especificar las localizaciones distintas al *classpath*, como en:

```
rmic c15.rmi.TiempoPerfecto
```

No es necesario estar en el directorio que contenga **TiempoPerfecto.class** cuando se ejecute este comando, si bien los resultados se colocarán en el directorio actual.

Cuando **rmic** se ejecuta con éxito, se tendrán dos nuevas clases en el directorio:

```
TiempoPerfecto_Stub.class
TiempoPerfecto_Skel.class
```

correspondientes al *stub* y al *skeleton*. Ahora, ya estamos listos para que el cliente y el servidor se intercomuniquen.

Utilizar el objeto remoto

Toda la motivación de RMI es simplificar el uso de objetos remotos. Lo único extra que hay que hacer en el programa cliente es buscar y capturar la interfaz remota desde el servidor. A partir de ese momento, no hay más que programación Java ordinaria: envío de mensajes a objetos. He aquí el programa que hace uso de **TiempoPerfecto**:

```
//: c15:rmi:MostrarTiempoPerfecto.java
// Usa el objeto remoto TiempoPerfecto.
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;

public class MostrarTiempoPerfecto {
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
      new RMISecurityManager());
    ITiempoPerfecto t =
      (ITiempoPerfecto)Naming.lookup(
        "//pepe:2005/TiempoPerfecto");
    for(int i = 0; i < 10; i++)
      System.out.println("TiempoPerfecto = " +
        t.obtenerTiempoPerfecto());
  }
} ///:~
```

La cadena de caracteres ID es la misma que la que se usó para registrar el objeto con **Naming**, y la primera parte representa al URL y al número de puerto. Dado que se está usando una URL también se puede especificar una máquina en Internet.

Lo que se devuelve de **Naming.lookup( )** hay que convertirlo a la interfaz remota, *no* a la clase. Si se usa la clase en su lugar, se obtendrá una excepción.

En la llamada a método:

```
t.obtenerTiempoPerfecto()
```

puede verse que una vez que se tiene una referencia al objeto remoto, la programación con él no difiere de la programación con un objeto local (con una diferencia: los métodos remotos lanzan **RemoteException**).

CORBA

En aplicaciones distribuidas grandes, las necesidades pueden no verse satisfechas con estos enfoques que acabamos de describir. Por ejemplo, uno podría querer interactuar con almacenes de datos antiguos, o podría necesitar servicios de un objeto servidor independientemente de su localizaron física. Estas situaciones requieren de algún tipo de *Remote Procedure Call* (RPC), y posiblemente de independencia del lenguaje. Es aquí donde CORBA puede ser útil.

CORBA no es un aspecto del lenguaje; es una tecnología de integración. Es una especificación que los fabricantes pueden seguir para implementar productos de integración compatibles con CORBA. Éste es parte del esfuerzo de la OMG para definir un marco estándar para interoperabilidad de objetos distribuidos independientemente del lenguaje.

CORBA proporciona la habilidad de construir llamadas a procedimientos remotos en objetos Java y no Java, y de interactuar con sistemas antiguos de forma independiente de la localización. Java añade soporte a redes y un lenguaje orientado a objetos perfecto para construir aplicaciones gráficas o no. El modelo de objetos de Java y el de la OMG se corresponden perfectamente entre sí; por ejemplo, ambos implementan el concepto de interfaz y un modelo de objetos de referencia.

Fundamentos de CORBA

A la especificación de la interoperabilidad entre objetos desarrollada por la OMG se le suele denominar la Arquitectura de Gestión de Objetos (OMA, *Object Management Architecture*). La OMA define dos conceptos: el *Core Object Model* y la *OMA Reference Architecture*. El primero establece los conceptos básicos de un objeto, interfaz, operación, etc. (CORBA es un refinamiento del *Core Object Model*). La OMA *Reference Architecture* define una infraestructura de servicios y mecanismos subyacentes que permiten interoperar a los objetos. Incluye el *Object Request Broker* (ORB), *Object Services* (conocidos también como *CORBA services*) y facilidades generales.

El ORB es el canal de comunicación a través del cual unos objetos pueden solicitar servicios a otros, independientemente de su localización física. Esto significa que lo que parece una llamada a un mé-

todo en el código cliente es, de hecho, una operación compleja. En primer lugar, debe existir una conexión con el objeto servidor, y para crear la conexión el ORB debe saber dónde reside el código que implementa ese servidor. Una vez establecida la conexión, hay que pasar los parámetros del método, por ejemplo, convertidos en un flujo binario que se envía a través de la red. También hay que enviar otra información como el nombre de la máquina servidora, el proceso servidor y la identidad del objeto servidor dentro de ese proceso. Finalmente, esta información se envía a través de un protocolo de bajo nivel, se decodifica en el lado servidor y se ejecuta la llamada. El ORB oculta toda esta complejidad al programador y hace la operación casi tan simple como llamar a un método de un objeto local. No hay ninguna especificación que indique cómo debería implementarse un nucleo ORB, pero para proporcionar compatibilidad básica entre los ORB de los diferentes vendedores, la OMG define un conjunto de servicios accesibles a través de interfaces estándar.

Lenguaje de Definición de Interfaces CORBA (IDL)

CORBA está diseñado para lograr la transparencia del lenguaje: un objeto cliente puede invocar a métodos de un objeto servidor de distinta clase, independientemente del lenguaje en que estén implementados. Por supuesto, el objeto cliente debe conocer los nombres y signaturas de los métodos que expone el objeto servidor. Es aquí donde interviene el IDL. El CORBA IDL es una forma independiente del lenguaje de especificar tipos de datos, atributos, operaciones, interfaces y demás. La sintaxis IDL es semejante a la de C++ o Java. La tabla siguiente muestra la correspondencia entre algunos de los conceptos comunes a los tres lenguajes, que pueden especificarse mediante CORBA IDL:

CORBA IDL	**Java**	**C++**
Módulo	Paquete	Espacio de nombre
Interfaz	Interfaz	Clase abstracta pura
Método	Método	Función miembro

También se soporta el concepto de herencia, utilizando el operador "dos puntos" como en C++. El programador escribe una descripción IDL de los atributos, métodos e interfaces implementados y usados por el servidor y los clientes. Después, se compila el IDL mediante un compilador IDL/Java proporcionado por un fabricante, que lee el fuente IDL y genera código Java.

El compilador IDL es una herramienta extremadamente útil: no genera simplemente un código fuente Java equivalente al IDL, sino que también genera el código que se usará para pasar los parámetros a métodos y para hacer llamadas remotas. A estos códigos se les llama *stub* y *skeleton*, y están organizados en múltiples archivos fuente Java, siendo generalmente parte del mismo paquete Java.

El servicio de nombres

El servicio de nombres es uno de los servicios CORBA fundamentales. A un objeto CORBA se accede a través de una referencia, un fragmento de información que no tiene significado para un lector humano. Pero a las referencias pueden asignarse nombres o cadenas de caracteres definidas por el programador. A esta operación se le denomina *encadenar la referencia*, y uno de los componentes de OMA, el Servicio de Nombres, se dedica exclusivamente a hacer conversiones y correspondencias cadena y objeto-a-cadena. Puesto que el servicio de nombres actúa como un directorio de teléfonos y, tanto servidores como clientes pueden consultarlo y manipularlo, se ejecuta como un proceso aparte. A la creación de una correspondencia objeto-a-cadena se le denomina vinculación de un objeto y a la eliminación de esta correspondencia de le denomina *desvinculación*. A la obtención de una referencia de un objeto pasando un String se le denomina *resolución de un nombre*.

Por ejemplo, al arrancar, una aplicación servidora podría crear un objeto servidor, establecer una correspondencia en el servicio de nombres y esperar después a que los clientes hagan peticiones. Un cliente obtiene primero una referencia al objeto servidor, resuelve el string, y después puede hacer llamadas al servidor haciendo uso de la referencia.

De nuevo, la especificación del servicio de nombres es parte de CORBA, pero la aplicación que lo proporciona está implementada por el fabricante del ORB. La forma de acceder a la funcionalidad del Servicio de Nombres puede variar de un fabricante a otro.

Un ejemplo

El código que se muestra aquí no será muy elaborado ya que distintos ORB tienen distintas formas de acceder a servicios CORBA, por lo que los ejemplos son específicos de los fabricantes. (El ejemplo de debajo usa JavaIDL, un producto gratuito de Sun que viene con un ORB ligero, un servicio de nombres, y un compilador de IDL a Java.) Además, dado que Java es un lenguaje joven y en evolución, no todas las facetas de CORBA están presentes en los distintos productos Java/CORBA.

Queremos implementar un servidor, en ejecución en alguna máquina, al que se pueda preguntar la hora exacta. También se desea implementar un cliente que pregunte por la hora exacta. En este caso, se implementarán ambos programas en Java, pero también se podrían haber usado dos lenguajes distintos (lo cual ocurre a menudo en situaciones reales).

Escribir el IDL fuente

El primer paso es escribir una descripción IDL de los servicios proporcionados. Esto lo suele hacer el programador del servidor, que es después libre de implementar el servidor en cualquier lenguaje en el que exista un compilador CORBA IDL. El archivo IDL se distribuye al programador de la parte cliente y se convierte en el puente entre los distintos lenguajes.

El ejemplo de debajo muestra la descripción IDL de nuestro servidor **TiempoExacto**:

```
//: c15:corba:TiempoExacto.idl
//# Hay que instalar el idltojava.exe de
//# java.sun.com y configurarlo para usar el
```

```
//# preprocesador C local para que compile
//# este archivo. Ver documentos de java.sun.com.
module tiemporemoto {
    interface TiempoExacto {
        string getTime();
    };
}; ///:~
```

Ésta es la declaración de una interfaz **TiempoExacto** de dentro del espacio de nombres **tiemporemoto**. La interfaz está formada por un único método que devuelve la hora actual en formato **string**.

Crear stubs y skeletons

El segundo paso es compilar el IDL para crear el código *stub* y el *skeleton* de Java que se usará para implementar el cliente y el servidor. La herramienta que viene con el producto JavaIDL es **idltojava**:

```
idltojava tiemporemoto.idl
```

Esto generará automáticamente el código, tanto para el *stub* como para el *skeleton*. **Idltojava** genera un **package** Java cuyo nombre se basa en el del módulo IDL, **tiemporemoto**, y los archivos Java generados se ponen en el subdirectorio **tiemporemoto**. El *skeleton* es **_TiempoExactoImplBase.java**, que se usará par implementar el objeto servidor, y **_TiempoExactoStub.java** se usará para el cliente. Hay representaciones Java de la interfaz IDL en **TiempoExacto.java** y un par de otros archivos de soporte que se usan, por ejemplo, para facilitar el acceso a operaciones de servicio de nombres.

Implementar el servidor y el cliente

Debajo puede verse el código correspondiente al lado servidor. La implementación del objeto servidor está en la clase **ServidorTiempoExacto**. El **ServidorTiempoRemoto** es la aplicación que crea un objeto servidor, lo registra en el ORB, le da un nombre a la referencia al objeto, y después queda a la espera de peticiones del cliente.

```
//: c15:corba:ServidorTiempoRemoto.java
import tiemporemoto.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Implementación del objeto servidor
class ServidorTiempoExacto extends _TiempoExactoImplBase {
  public String obtenerTiempo(){
    return DateFormat.
        getTimeInstance(DateFormat.FULL).
          format(new Date(
              System.currentTimeMillis()));
```

```
    }
}

// Implementación de la aplicación remota
public class ServidorTiempoRemoto {
  // Lanza excepciones a la consola:
  public static void main(String[] args)
  throws Exception {
    // Creación e inicialización del ORB:
    ORB orb = ORB.init(args, null);
    // Crear el objeto servidor y registrarlo:
    ServidorTiempoExacto refObjServidorTiempo =
      new ServidorTiempoExacto();
    orb.connect(refObjServidorTiempo);
    // Conseguir el contexto de raíz de los nombres:
    org.omg.CORBA.Object refObj =
      orb.resolve_initial_references(
        "NameService");
    NamingContext ncRef =
      NamingContextHelper.narrow(refObj);
    // Asignar un string a la
    // referencia a objetos (ubicación):
    NameComponent nc =
      new NameComponent("TiempoExacto", "");
    NameComponent[] ruta = { nc };
    ncRef.rebind(ruta, refObjServidorTiempo);
    // Esperar peticiones de clientes:
    java.lang.Object sinc =
      new java.lang.Object();
    synchronized(sinc){
      sinc.wait();
    }
  }
} ///:~
```

Como puede verse, implementar el objeto servidor es simple; es una clase normal Java que se hereda del código *skeleton* generado por el compilador IDL. Las cosas se complican cuando hay que interactuar con el ORB y otros servicios CORBA.

Algunos servicios CORBA

He aquí una breve descripción de lo que está haciendo el código JavaIDL (ignorando principalmente la parte del código CORBA dependiente del vendedor). La primera línea del método **main( )** arranca el ORB, y por supuesto, esto se debe a que el objeto servidor necesitará interactuar con él. Justo después de la inicialización del ORB se crea un objeto servidor. De hecho, el término correcto sería un *objeto sirviente transitorio*: un objeto que recibe peticiones de clientes, cuya longevidad

es la misma que la del proceso que lo crea. Una vez que se ha creado el objeto sirviente transitorio, se registra en el ORB, lo que significa que el ORB sabe de su existencia y que puede ahora dirigirle peticiones.

Hasta este punto, todo lo que tenemos es **RefObjServidorTiempo**, una referencia a objetos conocida sólo dentro del proceso servidor actual. El siguiente paso será asignarle un nombre en forma de string a este objeto sirviente; los clientes usarán ese nombre para localizarlo. Esta operación se logra haciendo uso del Servicio de Nombres. Primero, necesitamos una referencia a objetos al Servicio de Nombres; la llamada a **resolve_initial_references( )** toma la referencia al objeto pasado a string del Servicio de Nombres, que es "NameService", en JavaIDL, y devuelve una referencia al objeto. Ésta se convierte a una referencia **NamingContext** específica usando el método **narrow( )**. Ahora ya pueden usarse los servicios de nombres.

Para vincular el objeto sirviente con una referencia a objetos pasada a string, primero se crea un objeto **NameComponent**, inicializado con "TiempoExacto", la cadena de caracteres que se desea vincular al objeto sirviente. Después, haciendo uso del método **rebind( )**, se vincula la referencia pasada a string a la referencia al objeto. Se usa **rebind( )** para asignar una referencia, incluso si ésta ya existe, mientras que **bind( )** provoca una excepción si la referencia ya existe. En CORBA un nombre está formado por varios NameContexts —por ello se usa un array para vincular el nombre a la referencia al objeto.

Finalmente, el objeto sirviente ya está listo para ser usado por los clientes. En este punto, el proceso servidor entra en un estado de espera. De nuevo, esto se debe a que es un sirviente transitorio, por lo que su longevidad podría estar confinada al proceso servidor. JavaIDL no soporta actualmente objetos persistentes —objetos que sobreviven a la ejecución del proceso que los crea.

Ahora que ya se tiene una idea de lo que está haciendo el código servidor, echemos un vistazo al código cliente:

```
//: c15:corba:ClienteTiempoRemoto.java
import tiemporemoto.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class ClienteTiempoRemoto {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws Exception {
    // Creación e inicialización del ORB:
    ORB orb = ORB.init(args, null);
    // Obtener el contexto de nombrado raíz:
    org.omg.CORBA.Object refObj =
      orb.resolve_initial_references(
        "NameService");
    NamingContext ncRef =
      NamingContextHelper.narrow(refObj);
    // Lograr (resolver) la referencia del objeto
```

```
      // pasado a String para el servidor de hora:
      NameComponent nc =
         new NameComponent("TiempoExacto", "");
      NameComponent[] ruta = { nc };
      TiempoExacto refObjTiempo =
         TiempoExactoHelper.narrow(
            ncRef.resolve(ruta));
      // Hacer peticiones al objeto servidor:
      String tiempoExacto = refObjTiempo.obtenerTiempo();
      System.out.println(tiempoExacto);
   }
} ///:~
```

Las primeras líneas hacen los mismo que en el proceso servidor: se inicializa el ORB y se resuelve una referencia al servidor del Servicio de Nombres. Después se necesita una referencia a un objeto para el objeto sirviente, por lo que le pasamos una referencia a objeto pasada a String al método **resolve( )**, y convertimos el resultado en una referencia a la interfaz **TiempoExacto** usando el método **narrow( )**. Finalmente, invocamos a **obtenerTiempo( )**.

Activar el proceso de servicio de nombres

Finalmente, tenemos una aplicación servidora y una aplicación cliente listas para interoperar. Hemos visto que ambas necesitan el servicio de nombres para vincular y resolver referencias a objetos pasadas a String. Hay que arrancar el proceso de servicio de nombres antes de ejecutar o el servidor o el cliente. En JavaIDL el servicio de nombres es una aplicación Java que viene con el paquete del producto, pero puede ser distinta en cada producto. El servicio de nombres JavaIDL se ejecuta dentro de una instancia de la JVM y escucha por defecto al puerto 900 de red.

Activar el servidor y el cliente

Ahora ya se pueden arrancar las aplicaciones servidor y cliente (en este orden, pues el servidor es temporal). Si todo se configura bien, lo que se logra es una única línea de salida en la ventana de consola del cliente, con la hora actual. Por supuesto, esto puede que de por sí no sea muy excitante, pero habría que tener algo en cuenta: incluso si están en la misma máquina física, la aplicación cliente y la servidora se están ejecutando en máquinas virtuales distintas y se pueden comunicar vía una capa de integración subyacente, el ORB y el Servicio de Nombres.

Éste es un ejemplo simple, diseñado para trabajar sin red, pero un ORB suele estar configurado para buscar transparencia de la localización. Cuando el servidor y el cliente están en máquinas distintas, el ORB puede resolver referencias pasadas a Strings, remotas, utilizando un componente denominado el *Repositorio de implementaciones*. Aunque éste es parte CORBA, casi no hay ninguna especificación relativa al mismo, por lo que varía de fabricante en fabricante.

Como puede verse, hay mucho más de CORBA que lo que se ha cubierto hasta aquí, pero uno ya debería haber captado la idea básica. Si se desea más información sobre CORBA un buen punto de partida es el sitio web de OMG, en *http://www.omg.org*. Ahí hay documentación, *white papers*, procedimientos y referencias a otras fuentes y productos relacionados con CORBA.

Applets de Java y CORBA

Los *applets* de Java pueden actuar como clientes CORBA. De esta forma, un *applet* puede acceder a información remota y a servicios expuestos como objetos CORBA. Pero un *applet* sólo se puede conectar con el servidor desde el que fue descargado, por lo que todos los objetos CORBA con los que interactúe el *applet* deben estar en ese servidor. Esto es lo contrario a lo que intenta hacer CORBA: ofrecer una transparencia total respecto a la localización.

Éste es un aspecto de la seguridad de la red. Si uno está en una Intranet, una solución es disminuir las restricciones de seguridad en el navegador. O establecer una política de *firewall* para la conexión con servidores externos.

Algunos productos Java ORB ofrecen soluciones propietarias a este problema. Por ejemplo, algunos implementan lo que se denomina *Tunneling HTTP*, mientras que otros tienen sus propias facetas *firewall*.

Éste es un tema demasiado complejo como para cubrirlo en un apéndice, pero definitivamente es algo de lo que habría que estar pendientes.

CORBA frente a RMI

Se ha visto que una de las facetas más importantes de CORBA es el soporte RPC, que permite a los objetos locales invocar a métodos de objetos remotos. Por supuesto, ya hay una faceta nativa Java que hace exactamente lo mismo: RMI (véase Capítulo 15). Mientras que RMI hace posibles RPC entre objetos Java, CORBA las hace posibles entre objetos implementados en cualquier lenguaje. La diferencia es, pues, enorme.

Sin embargo, RMI puede usarse para invocar a servicios en código remoto no Java. Todo lo que se necesita es algún tipo de envoltorio de objeto Java en torno al código no Java del lado servidor. El objeto envoltorio se conecta externamente a clientes Java vía RMI, e internamente se conecta al código no Java usando una de las técnicas mostradas anteriormente, como JNI o J/Direct.

Este enfoque requiere la escritura de algún tipo de capa de integración, que es exactamente lo que hace CORBA automáticamente, pero de esta forma no se necesitaría un ORB elaborado por un tercero.

Enterprise JavaBeans

Supóngase[6] que hay que desarrollar una aplicación multicapa para visualizar y actualizar registros de una base de datos a través de una interfaz web. Se puede escribir un aplicación de base de datos usando JDBC, una interfaz web usando JSP/servlets, y un sistema distribuido usando CORBA/RMI.

---

[6] Sección a la que ayudó Robert Castaneda, ayudado a su vez por Dave Bartlett.

Pero, ¿qué consideraciones extra hay que hacer al desarrollar un sistema de objetos distribuido además de simplemente conocer las API? He aquí los distintos aspectos:

Rendimiento: los objetos distribuidos que uno crea deben ejecutarse con buen rendimiento, pues potencialmente podrían servir a muchos clientes simultáneamente. Habrá que usar técnicas de optimización como la captura y organización de recursos como conexiones a base de datos. También habrá que gestionar el ciclo de vida de los objetos distribuidos.

Escalabilidad: los objetos distribuidos deben ser también escalables. La escalabilidad en una aplicación distribuida significa que se pueda incrementar el número de instancias de los objetos distribuidos, trasladándose a máquinas adicionales sin necesidad de modificar ningún código.

Seguridad: un objeto distribuido suele tener que gestionar la autorización de los clientes que lo acceden. Idealmente, se pueden añadir nuevos usuarios y roles al mismo sin tener que recompilar.

Transacciones distribuidas: un objeto distribuido debería ser capaz de referenciar a transacciones distribuidas de forma transparente. Por ejemplo, si se está trabajando con dos base de datos separadas, habría que poder actualizarlas simultáneamente dentro de la misma transacción, o deshacer una transacción si no se satisface determinado criterio.

Reusabilidad: el objeto distribuido ideal puede moverse sin esfuerzo a otro servidor de aplicaciones de otro vendedor. Sería genial si se pudiera revender un componente objeto distribuido sin tener que hacerle modificaciones especiales, o comprar un componente de un tercero y usarlo sin tener que recompilarlo ni reescribirlo.

Disponibilidad: si una de las máquinas del sistema se cae, los clientes deberían dirigirse automáticamente a copias de respaldo de esos objetos, residentes en otras máquinas.

Estas consideraciones, además del problema de negocio que se trata de solucionar, pueden hacer un proyecto inabordable. Sin embargo, todos los aspectos *excepto* los del problema de negocio son redundantes —hay que reinventar soluciones para cada aplicación de negocio distribuida.

Sun, junto con otros fabricantes de objetos distribuidos líderes, se dio cuenta de que antes o después todo equipo de desarrollo estaría reinventando estas soluciones particulares, por lo que crearon la especificación de los *Enterprise JavaBeans* (EJB). Esta especificación describe un modelo de componentes del lado servidor que toma en consideración todos los aspectos mencionados utilizando un enfoque estándar que permite a los desarrolladores crear componentes de negocio denominados EJBs, que se aíslan del código "pesado" de bajo nivel y se enfocan sólo en proporcionar lógica de negocio. Dado que los EJB están definidos de forma estándar, pueden ser independientes del fabricante.

JavaBeans frente a EJB

Debido a la similitud de sus nombres, hay mucha confusión en la relación entre el modelo de componentes JavaBeans y la especificación de los *Enterprise JavaBeans*. Mientras que, tanto unos como otros comparten los mismos objetivos de promocionar la reutilización y portabilidad del código Java entre las herramientas de desarrollo y diseminación con el uso de patrones de diseño estándares,

los motivos que subyacen tras cada especificación están orientados a solucionar problemas diferentes.

Los estándares definidos en el modelo de comportamiento de los JavaBeans están diseñados para crear componentes reusables que suelen usarse en herramientas de desarrollo IDE y comúnmente, aunque no exclusivamente, en componentes visuales.

La especificación de los JavaBeans define un modelo de componentes para desarrollar código Java del lado servidor. Dado que los EJBs pueden ejecutarse potencialmente en distintas plataformas de la parte servidor —incluyendo *servidores* que no tienen presentación visuales— un EJB no puede hacer uso de las bibliotecas gráficas, como la AWT o Swing.

La especificación EJB

La especificación de *Enterprise JavaBeans* describe un modelo de componentes del lado servidor. Define seis papeles que se usan para llevar a cabo las tareas de desarrollo y distribución además de definir los componentes del sistema. Estos papeles se usan en el desarrollo, distribución, y ejecución de un sistema distribuido. Los fabricantes, administradores y desarrolladores van jugando los distintos papeles, para permitir la división del conocimiento técnico y del dominio. El fabricante proporciona un marco de trabajo de sonido en su parte técnica, y los desarrolladores crean componentes específicos del dominio; por ejemplo, un componente "contable". Una misma parte puede llevar a cabo uno o varios papeles. Los papeles definidos en la especificación de EJB se resumen en la tabla siguiente:

Papel	**Responsabilidad**
Suministrador de *Enterprise Bean*	El desarrollador responsable de crear componentes EJB reusables. Estos componentes se empaquetan en un archivo jar especial (archivo ejb-jar).
Ensamblador de aplicaciones	Crea y ensambla aplicaciones a partir de una colección de archivos ejb-jar. Incluye la escritura de aplicaciones que usan la colección de EJB (por ejemplo, servlets, JSP, Swing, etc.,etc.).
Distribuidor	Toma la colección de archivos ejb-jar del ensamblador y/o suministrador de Beans y los distribuye en un entorno de tiempo de ejecución: uno o más contenedores EJB.
Proveedor de contenedores/ servidores de EJB	Proporciona un entorno de tiempo de ejecución y herramientas que se usan para distribuir, administrar y ejecutar componentes EJB.
Administrador del sistema	Gestiona los distintos componentes y servicios, de forma que estén configurados e interactúen correctamente, además de asegurar que el sistema esté activo y en ejecución.

Componentes EJB

Los componentes EJB son elementos de lógica de negocio reutilizable que se adhieren a estándares estrictos y patrones de diseño definidos en la especificación EJB. Esto permite la portabilidad de los componentes. También permite poder llevar a cabo otros servicios —como la seguridad, la gestión de cachés, y las transacciones distribuidas— por parte de los componentes. El responsable del desarrollo de componentes EJB es el *Enterprise Bean Provider*.

Contenedor & Servidor EJB

El *Contenedor EJB* es un entorno de tiempo de ejecución que contiene y ejecuta componentes EJB y les proporciona un conjunto de servicios estándares. Las responsabilidades del contenedor de EJB están definidas de forma clara en la especificación, en aras de lograr neutralidad respecto del fabricante. El contenedor de EJB proporciona el "peso pesado" de bajo nivel del EJB, incluyendo transacciones distribuidas, seguridad, gestión del ciclo de vida de los *beans*, gestión de caché, hilado y gestión de sesiones. El proveedor de contenedor EJB es el responsable de su provisión.

Un *Servidor EJB* se define como un servidor de aplicación que contiene y ejecuta uno o más contenedores de EJB. El Proveedor de Servicios EJB es responsable de proporcionar un Servidor EJB. Generalmente se puede asumir que el Contenedor de EJBs y el Servidor de EJBs son el mismo.

Interfaz Java para Nombres y Directorios (JNDI)[7]

Interfaz usado en los *Enterprise JavaBeans* como el servicio de nombres para los componentes EJB de la red y para otros servicios de contenedores, como las transacciones. JNDI establece una correspondencia nuy estricta con otros estándares de nombres y directorios como *CORBA CosNaming* y puede implementarse, realmente, como un envoltorio realmente de éstos.

Java Transaction API / Java Transaction Service (JTA/JTS)

JTA/JTS se usa en las *Enterprise JavaBeans* como el API transaccional. Un proveedor de *Enterprise Beans* puede usar JTS para crear código de transacción, aunque el contenedor EJB suele implementar transacciones EJB de parte de los componentes EJB. El distribuidor puede definir los atributos transaccionales de un componente EJB en tiempo de distribución. El Contenedor de EJB es el responsable de gestionar la transacción, sea local o distribuida. La especificación JTS es la correspondencia Java al CORBA OTS (*Object Transaction Service*).

CORBA y RMI/IIOP

La especificación EJB define la interoperabilidad con CORBA a través de la compatibilidad con protocolos CORBA. Esto se logra estableciendo una correspondencia entre servicios EJB como JTS y JNDI con los servicios CORBA correspondientes, y la implementación de RMI sobre el protocolo IIOP de CORBA.

---

[7] N. del traductor: *Java Naming and Directory Interface.*

El uso de CORBA y RMI/IIOP en *Enterprise JavaBeans* está implementado en el contenedor de EJB y es el responsable del proveedor de contenedores EJB. El uso de CORBA y de RMI/IIOP sobre el contenedor de EJB está oculto desde el propio componente EJB. Esto significa que el proveedor de *Enterprise Beans* puede escribir su componente EJB y distribuirlo a cualquier contenedor de EJB sin que importe qué protocolo de comunicación se esté utilizando.

Las partes de un componente EJB

Un EJB está formado por varias piezas, incluyendo el propio Bean, la implementación de algunas interfaces, y un archivo de información. Todo se empaqueta junto en un archivo jar especial.

Enterprise Bean

El Enterprise Bean es una clase Java que desarrolla el *Enterprise Bean Provider.* Implementa una interfaz de *Enterprise Bean* y proporciona la implementación de los métodos de negocio que va a llevar a cabo el componente. La clase no implementa ninguna autorización, autentificación, multihilo o código transaccional.

Interfaz local

Todo *Enterprise Bean* que se cree debe tener su interfaz local asociado. Esta interfaz se usa como fábrica del EJB. Los clientes lo usan para encontrar una instancia del EJB o crear una nueva.

Interfaz remota

Se trata de una interfaz Java que refleja los métodos del *Enterprise Bean* que se desea exponer al mundo exterior. Esta interfaz juega un papel similar al de una interfaz CORBA IDL.

Descriptor de distribución

El descriptor de distribución es un archivo XML que contiene información sobre el EJB. El uso de XML permite al distribuidor cambiar fácilmente los atributos del EJB. Los atributos configurables definidos en el descriptor de distribución incluyen:

- Los nombres de interfaz Local y Remota requeridos por el EJB.
- El nombre a publicar en el JNDI para la interfaz local del EJB.
- Atributos transaccionales para cada método de EJB.
- Listas de control de acceso para la autenticación.

Archivo EJB-Jar

Es un archivo jar normal que contiene el EJB, las interfaces local y remota, y el descriptor de distribución.

Funcionamiento de un EJB

Una vez que se tiene un archivo EJB-Jar que contiene el Bean, las interfaces Local y Remota, y el descriptor de distribución, se pueden encajar todas las piezas y a la vez entender por qué se necesitan las interfaces Local y Remota, y cómo los usa el contendor de EJB.

El Contenedor implementa las interfaces Local y Remoto del archivo EJB-Jar. Como se mencionó anteriormente, la interfaz Local proporciona métodos para crear y localizar el EJB. Esto significa que el contenedor de EJB es responsable de la gestión del ciclo de vida del EJB. Este nivel de indirección permite que se den optimizaciones. Por ejemplo, 5 clientes podrían solicitar simultáneamente la creación de un EJB a través de una interfaz Local, y el contenedor de EJB respondería creando sólo un EJB y compartiéndolo entre los 5 clientes. Esto se logra a través de la interfaz Remota, que también está implementado por el Contenedor de EJB. El objeto Remoto implementado juega el papel de objeto *proxy* al EJB.

Todas las llamadas al EJB pasan por el *proxy* a través del contenedor de EJB vía las interfaces Local y Remota. Esta indirección es la razón por la que el contenedor de EJB puede controlar la seguridad y el comportamiento transaccional.

Tipos de EJB

La especificación de *Enterprise JavaBeans* define distintos tipos de EJB con características y comportamientos diferentes. En la especificación se han definido dos categorías de EJB: *Beans* de sesión y *Beans* de entidad, y cada categoría tiene sus propias variantes.

Beans de Sesión

Se usan para representar minúculos casos de uso o flujo de trabajo por parte de un cliente. Representan operaciones sobre información persistente, pero no a la información persistente en sí. Hay dos tipos de Beans de sesión: los que no tienen estado y los que lo tienen. Todos los Beans de sesión deben implementar la interfaz **javax.ejb.SessionBean**. El contenedor de EJB gobierna la vida de un Bean de Sesión.

Beans de Sesión sin estado: son el tipo de componente EJB más simple de implementar. No mantienen ningún estado conversacional con clientes entre invocaciones a métodos por lo que son fácilmente reusables en el lado servidor, y dado que pueden ser almacenados en cachés, se escalan bien bajo demanda. Cuando se usen, hay que almacenar toda información de estado fuera del EJB.

Beans de Sesión con estado: mantienen el estado entre invocaciones. Tienen una correspondencia lógica de uno a uno con el cliente y pueden mantener el estado entre ellas. El responsable de su almacenamiento y paso a caché es el Contenedor de EJB, que lo logra mediante su *Pasivación* y *Activación*. Si el contenedor de EJB falla, se perdería toda la información de los Beans de sesión con estado. Algunos Contenedores de EJB avanzados proporcionan posibilidad de recuperación para estos beans.

Beans de entidad

Son componentes que representan información persistente y comportamiento de estos datos. Estos Beans los pueden compartir múltiples clientes, de la misma forma que puede compartirse la información de una base de datos. El Contenedor de EJB es el responsable de gestionar en la caché los Beans de Entidad, y de mantener su integridad. La vida de estos beans suele ir más allá de la vida del Contenedor de EJB por lo que si éste falla, se espera que el Bean de Entidad siga disponible cuando el contenedor vuelva a estar activo.

Hay dos tipos de Beans de Entidad: los que tienen Persistencia Gestionada por el Contenedor y los de Persistencia Gestionada por el Bean.

Persistencia Gestionada por el Contenedor (CMP o ***Container Managed Persistence*****).** Un Bean de Entidad CMP tiene su persistencia implementada en el Contenedor de EJB. Mediante atributos especificados en el descriptor de distribución, el Contenedor de EJB establecerá correspondencias entre los atributos del Bean de Entidad y algún almacenamiento persistente (generalmente —aunque no siempre— una base de datos). CMP reduce el tiempo de desarrollo del EJB, además de reducir dramáticamente la cantidad de código necesaria.

Persistencia Gestionada por el Bean (BMP o ***Bean Managed Persistence*****).** Un Bean de entidad BMP tiene su persistencia implementada por el proveedor de Enterprise Beans. Éste es responsable de implementar la lógica necesaria para crear un nuevo EJB, actualizar algunos atributos de los EJB, borrar un EJB, y encontrar un EJB a partir de un almacén persistente. Esto suele implicar la escritura de código JDBC para interactuar con una base de datos u otro almacenamiento persistente. Con BMP, el desarrollado tiene control total sobre la gestión de la persistencia del Bean de Entidad.

BMP también proporciona flexibilidad allí donde puede no es posible implementar un CMP. Por ejemplo, si se deseara crear un EJB que envolviera algún código existente en un servidor, se podría escribir la persistencia usando CORBA.

Desarrollar un EJB

Como ejemplo, se implementará como componente EJB el ejemplo "Tiempo Perfecto" de la sección RMI anterior. El ejemplo será un sencillo Bean de Sesión sin Estado.

Como se mencionó anteriormente, los componentes EJB consisten en al menos una clase (el EJB) y dos interfaces: el Remoto y el Local. Cuando se crea una interfaz remota para un EJB, hay que seguir las siguientes directrices:

1. La interfaz Remota debe ser **public**.
2. La interfaz Remota debe extender a la interfaz **javax.ejb.EJBObject.**
3. Cada método de la interfaz Remota debe declarar **java.rmi.RemoteException** en su cláusula **throws** además de cualquier excepción específica de la aplicación.

4. Cualquier objeto que se pase como parámetro o valor de retorno (bien directamente o embebido en algún objeto local) debe ser un tipo de datos RMI-IIOP válido (incluyendo otros objetos EJB).

He aquí una interfaz remota simple para el EJB TiempoPerfecto:

```
//: c15:ejb:TiempoPerfecto.java
//# Para compilar este archivo hay que instalar
//# la J2EE Java Enterprise Edition de
//# java.sun.com y añadir j2ee.jar al
//# CLASSPATH. Ver detalles en java.sun.com.
// Interfaz Remota de BeanTiempoPerfecto
import java.rmi.*;
import javax.ejb.*;

public interface TiempoPerfecto extends EJBObject {
  public long obtenerTiempoPerfecto()
    throws RemoteException;
} ///:~
```

La interfaz Local es la fábrica en la que se creará el componente. Puede definir métodos *create*, para crear instancias de EJB, o métodos *finder* que localizan EJB existentes y se usan sólo para Beans de entidad. Cuando se crea una interfaz Local para un EJB hay que seguir estas directrices:

1. La interfaz Local debe ser **public**.

2. La interfaz Local debe extender el interfaz **javax.ejb.EJBHome**.

3. Cada método *create* de la interfaz Local debe declarar **java.rmi.RemoteException** en su cláusula **throws** además de una **javax.ejb.CreateException**.

4. El valor de retorno de un método *create* debe ser una interfaz Remota.

5. El valor de retorno de un método *finder* (sólo para Beans de Entidad) debe ser una interfaz Remota o **java.util.Enumeration** o **java.util.Collection**.

6. Cualquier objeto que se pase como parámetro (bien directamente o embebido en un objeto local) debe ser un tipo de datos RMI-IIOP válido (esto incluye otros objetos EJB).

La convención estándar de nombres para las interfaces Locales es tomar el nombre de la interfaz Remoto y añadir "Home" al final. He aquí la interfaz Local para el EJB TiempoPerfecto:

```
//: c15:ejb:TiempoPerfectoHome.java
// Interfaz Local de BeanTiempoPerfecto.
import java.rmi.*;
import javax.ejb.*;

public interface TiempoPerfectoHome extends EJBHome {
```

```
   public TiempoPerfecto create()
      throws CreateException, RemoteException;
} ///:~
```

Ahora se puede implementar la lógica de negocio. Cuando se cree la clase de implementación del EJB, hay que seguir estas directrices, (nótese que debería consultarse la especificación de EJB para lograr una lista completa de las directrices de desarrollo de *Enterprise JavaBeans*):

1. La clase debe ser **public**.
2. La clase debe implementar una interfaz EJB (o **javax.ejb.SessionBean** o **javax.ejb.EntityBean**).
3. La clase debería definir métodos que se correspondan directamente con los métodos de la interfaz Remota. Nótese que la clase no implementa la interfaz Remota; hace de espejo de los métodos de la interfaz Remota pero *no* lanza **java.rmi.RemoteException**.
4. Definir uno o más métodos **ejbCreate( )** para inicializar el EJB.
5. El valor de retorno y los parámetros de todos los métodos deben ser tipos de datos RMI-IIOP válidos.

```
//: c15:ejb:BeanTiempoPerfecto.java
// Bean Simple de Sesión sin estado
// que devuelve la hora actual del sistema.
import java.rmi.*;
import javax.ejb.*;

public class BeanTiempoPerfecto
   implements SessionBean {
   private SessionContext contextoSesion;
   //devuelve el tiempo actual
   public long obtenerTiempoPerfecto() {
      return System.currentTimeMillis();
   }
   //  métodos EJB
   public void ejbCreate()
   throws CreateException {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void
   SetSessioncontext(SessionContext ctx) {
      TcontextoSesion = ctx;
   }
}///:~
```

Dado que este ejemplo es muy sencillo, los métodos EJB (**ejbCreate( )**, **ejbRemove( )**, **ejbActivate( )**, **ejbPassivate( )**) están todos vacíos. Estos métodos son invocados por el Contenedor de EJB y se usan para controlar el estado del componente. El método **setSesionContext( )** pasa un objeto **javax.ejb.SessionContext** que contiene información sobre el contexto del componente, como información de la transacción actual y de seguridad.

Tras crear el *Enterprise JavaBean*, tenemos que crear un descriptor de distribución. Éste es un archivo XML que describe el componente EJB, y que debería estar almacenado en un archivo denominado **ebj-jar.xml**.

```
//:! c15:ejb:ejb-jar.xml
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <description>Ejemplo del Capitulo 15</description>
  <display-name></display-name>
  <small-icon></small-icon>
  <large-icon></large-icon>
  <enterprise-beans>
    <session>
      <ejb-name>TiempoPerfecto</ejb-name>
      <home>TiempoPerfectoHome</home>
      <remote>TiempoPerfecto</remote>
      <ejb-class>BeanTiempoPerfecto</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <ejb-client-jar></ejb-client-jar>
</ejb-jar>
///:~
```

Pueden verse el componente, la interfaz Remota y la interfaz Local definidos dentro de la etiqueta <**session**> de este descriptor de distribución. Estos descriptores pueden ser generados automáticamente usando las herramientas de desarrollo de EJB.

Junto con el descriptor de distribución estándar **ejb-jar.xml**, la especificación EJB establece que cualquier etiqueta específica de un vendedor debería almacenarse en un archivo separado. Esto pretende lograr una alta portabilidad entre componentes y distintos tipos de contenedores de EJB.

Los archivos deben guardarse dentro de un Archivo Java estándar (JAR). Los descriptores de distribución deberían ubicarse dentro del subdirectorio **/META-INF** del archivo Jar.

Una vez que se ha definido el componente EJB dentro del descriptor de distribución, el distribuidor debería distribuir el componente EJB en el Contendor de EJB. Al escribir esto, el proceso de dis-

tribución era bastante "intensivo respecto a IGU" y específico de cada contendor de EJB individual, por lo que este repaso no documenta ese proceso. Todo contenedor de EJB, sin embargo, tendrá un proceso documentado para la distribución de un EJB.

Dado que un componente EJB es un objeto distribuido, el proceso de distribución también debería crear algunos *stubs* de cliente para invocar al componente. Estas clases deberían ubicarse en el *class-path* de la aplicación cliente. Dado que los componentes EJB pueden implementarse sobre RMI-IIOP (CORBA) o sobre RMI-JRMP, los *stubs* generados podrían variar entre contenedores de EJB; de cualquier manera, son clases generadas.

Cuando un programa cliente desea invocar a un EJB, debe buscar el componente EJB dentro de la JNDI y obtener una referencia al interfaz local del componente EJB. Esta interfaz se usa para crear una instancia del EJB.

En este ejemplo, el programa cliente es un programa Java simple, pero debería recordarse que podría ser simplemente un servlet, un JSP o incluso un objeto distribuido CORBA o RMI.

```
//: c15:ejb:ClienteTiempoPerfecto.java
// Programa cliente para BeanTiempoPerfecto

public class ClienteTiempoPerfecto {
public static void main(String[] args)
throws Exception {
  // Lograr un contexto JNDI usando
  // el servicio de nombre de JNDI:
  javax.naming.Context contexto =
    new javax.naming.InitialContext();
  // Buscar la interfaz local en
  // el JNDI Naming service:
  Object ref = contexto.lookup("tiempoPerfecto");
  // Convertir el objeto remoto la interfaz local:
  TiempoPerfectoHome home = (TiempoPerfectoHome)
    javax.rmi.PortableRemoteObject.narrow(
      ref, TiempoPerfectoHome.class);
  // Crear un objeto remoto para la interfaz local:
  TiempoPerfecto tp = home.create();
  // Invocar obtenerTiempoPerfecto()
  System.out.println(
    "EJB Tiempo Perfecto invocado, son las: " +
    tp.obtenerTiempoPerfecto() );
  }
} ///:~
```

La secuencia de este ejemplo se explica mediante comentarios. Nótese el uso del método **narrow( )** para llevar a cabo una conversión del objeto antes de llevar a cabo una conversión Java autentica.

Esto es muy semejante a lo que ocurre en CORBA. Nótese también que el objeto Home se convierte en una fábrica de objetos **TiempoPerfecto**.

Resumen de EJB

La especificación de *Enterprise JavaBeans* es un terrible paso adelante de cara a la estandarización y simplificación de la computación de objetos distribuidos. Es una de las piezas más importantes de la plataforma *Java 2 Enterprise Edition* (J2EE) y está recibiendo mucho soporte de la comunidad de objetos distribuidos. Actualmente hay muchas herramientas disponibles, y si no lo estarán en breve, para ayudar a acelerar el desarrollo de componentes EJB.

Este repaso sólo pretendía ser un breve *tour* por los EJB. Para más información sobre la especificación de EJB puede acudirse a la página oficial de los *Enterprise JavaBeans* en *http://java.sun.com/products/ejb* donde puede descargarse la última especificación y la implementación de referencia J2EE. Éstas pueden usarse para desarrollar y distribuir componentes EJB propios.

Jini: servicios distribuidos

Esta sección[8] da un repaso a la tecnología Jini de Sun Microsystems. Describe algunas de las ventajas e inconvenientes de Jini y muestra cómo su arquitectura ayuda a afrontar el nivel de abstracción en programación de sistemas distribuidos, convirtiendo de forma efectiva la programación en red en programación orientada a objetos.

Jini en contexto

Tradicionalmente, se han diseñado los sistemas operativos con la presunción de que un computador tendría un procesador, algo de memoria, y un disco. Cuando se arranca un computador lo primero que hace es buscar un disco. Si no lo encuentra, no funciona como computador. Cada vez más, sin embargo, están apareciendo computadores de otra guisa: como dispositivos embebidos que tienen un procesador, algo de memoria, y una conexión a la red —pero no disco. Lo primero que hace un teléfono móvil al arrancarlo, por ejemplo, es buscar una red de telefonía. Si no la encuentra, no puede funcionar como teléfono móvil. Esta tendencia en el entorno hardware, de centrados en el disco a centrados en la red, afectará a la forma que tenemos de organizar el software —y es aquí donde Jini cobra sentido.

Jini es un intento de replantear la arquitectura de los computadores, dada la importancia creciente de la red y la proliferación de procesadores en dispositivos sin unidad de disco. Estos dispositivos, que vendrán de muchos fabricantes distintos, necesitarán interactuar por una red. La red en sí será muy dinámica —regularmente se añadirán y eliminarán dispositivos y servicios. Jini proporciona mecanismos para permitir la adición, eliminación y búsqueda de directorios y servicios de forma sencilla a través de la red. Además, Jini proporciona un modelo de programación que facilita a los programadores hacer que sus dispositivos se comuniquen a través de la red.

---

[8] Esta sección se llevó a cabo con la ayuda de Bill Venners (http://www.artima.com).

Construido sobre Java, la serialización de objetos, y RMI (que permiten, entre todos, que los objetos se muevan por la red de máquina virtual en máquina virtual), Jini intenta extender los beneficios de la programación orientada a objetos a la red. En vez de pedir a fabricantes de dispositivos que se pongan de acuerdo en protocolos de red para que sus dispositivos interactúen, Jini habilita a los dispositivos para que se comuniquen mutuamente a través de interfaces con objetos.

¿Qué es Jini?

Jini es un conjunto de API y protocolos de red que pueden ayudar a construir y desplegar sistemas distribuidos organizados como *federaciones de servicios*. Un *servicio* puede ser cualquier cosa que resida en la red y que esté listo para llevar a cabo una función útil. Los dispositivos hardware, el software, los canales de comunicación —e incluso los propios seres humanos— podrían ser servicios. Una unidad de disco habilitada para Jini, por ejemplo, podría ofrecer un servicio de "almacenamiento". Una impresora habilitada para Jini podría ofrecer un servicio de "impresión". Por consiguiente, una federación de servicios es un conjunto de servicios disponible actualmente en la red, que un cliente (entendiendo por tal un programa, servicio o usuario) puede juntar para que le ayuden a lograr alguna meta.

Para llevar a cabo una tarea, un cliente lista la ayuda de varios servicios. Por ejemplo, un programa cliente podría mandar a un servidor fotos del almacén de imágenes de una cámara digital, descargar las fotos a un servicio de almacenamiento persistente ofrecido por una unidad de disco, y enviar una página de versiones del tamaño de una de las fotos al servicio de impresión de una impresora a color. En este ejemplo, el programa cliente construye un sistema distribuido que consiste en sí mismo, el servicio de almacenamiento de imágenes, el servicio de almacenamiento persistente, y el servicio de impresión en color. El cliente y los servicios de este sistema distribuido trabajan juntos para desempeñar una tarea: descargar y almacenar imágenes de una cámara digital e imprimir una página de copias diminutas.

La idea que hay tras la palabra *federación* es que la visión de Jini de la red no implique una autoridad de control central. Dado que no hay ningún servicio al mando, el conjunto de todos los servicios disponibles en la red conforman una federación —un grupo formado por iguales. En vez de una autoridad central, la infraestructura de tiempo de ejecución de Jini simplemente proporciona una forma para que los clientes y servicios se encuentren entre sí (vía un servicio de búsqueda que almacena un directorio de los servicios actualmente disponibles). Una vez que los servicios se localizan entre sí ya pueden funcionar. El cliente y sus servicios agrupados llevan a cabo su tarea independientemente de la infraestructura de tiempo de ejecución de Jini. Si el servicio de búsqueda de Jini se cae, cualquier sistema distribuido conformado vía el servicio de búsqueda antes de que se produjera el fallo puede continuar con su trabajo. Jini incluso incluye un protocolo de red que pueden usar los clientes para encontrar servicios en la ausencia de un servicio de búsqueda.

Cómo funciona Jini

Jini define una *infraestructura de tiempo de ejecución* que reside en la red y proporciona mecanismos que permiten a cada uno añadir, eliminar, localizar y acceder a servicios. La infraestructura de tiempo de ejecución reside en tres lugares: en servicios de búsqueda que residen en la red, en los proveedores de servicios (como los dispositivos habilitados para Jini), y en los clientes. Los *Servicios de bús-*

queda son el mecanismo de organización central de los sistemas basados en Jini. Cuando aparecen más servicios en la red, se registran a sí mismos con un servicio de búsqueda. Cuando los clientes desean localizar un servicio para ayudar con alguna tarea, consultan a un servicio de búsqueda.

La infraestructura de tiempo de ejecución usa un protocolo de nivel de red denominado *discovery*, y dos protocolos de nivel de red, llamados *join* y *lookup*. *Discovery* permite a los clientes y servicios localizar los servicios de búsqueda. *Join* permite a un servicio registrarse a sí mismo en un servicio de búsqueda. *Lookup* permite a un cliente preguntar por los servicios que pueden ayudar a lograr sus metas.

El proceso de discovery

Discovery funciona así: imagínese que se tiene una unidad de disco *habilitada para Jini* que ofrece un servicio de almacenamiento persistente. Tan pronto como se conecte el disco a la red, éste lanza en multidifusión un *anuncio de presencia* depositando un paquete de multidifusión en un puerto bien conocido. En este anuncio de presencia va incluida la dirección IP y el número de puerto en el que el servicio de búsqueda puede localizar la unidad de disco.

Los servicios de búsqueda monitorizan el puerto bien conocido en espera de paquetes de anuncio de presencia. Cuando un servicio de búsqueda recibe un anuncio de presencia, lo abre e inspecciona el paquete. Éste contiene información que permite al servicio de búsqueda determinar si debería o no contactar al emisor del paquete. En caso afirmativo, contacta con el emisor directamente haciendo una conexión TCP a la dirección IP y número de puerto extraídos del paquete. Usando RMI, el servicio de búsqueda envía un objeto denominado *registrador de servicios*, a través de la red, a la fuente del paquete. El propósito del servicio registrador de objetos es facilitar una comunicación más avanzada con el servicio de búsqueda. Al invocar a los métodos de este objeto, el emisor del paquete de anuncio puede unirse y buscar en el servicio de búsqueda. En el caso de la unidad de disco, el servicio de búsqueda haría una conexión TCP a la unidad de disco y le enviaría un objeto registrador de servicios, a través del cual registraría después la unidad de disco su servicio de almacenamiento persistente vía el proceso *join*.

El proceso join

Una vez que un proveedor de servicio tiene un objeto registrador de servicios, como resultado del *discovery*, está listo para hacer una *join* —convertirse en parte de la federación de servicios registrados en el servicio de búsqueda. Para ello, el proveedor del servicio invoca al método **register( )** del objeto registrador de servicios, pasándole como parámetro un objeto denominado un elemento de servicio, un conjunto de objetos que describen el servicio. El método **register( )** envía una copia del elemento de servicio al servicio de búsqueda, donde se almacena el elemento de servicio.

Una vez completada esta operación, el proveedor del servicio ha acabado el proceso *join*: su servicio ya está registrado en el servicio de búsqueda.

Este elemento de servicio es un contenedor de varios objetos, incluyendo uno llamado un *objeto de servicio*, que pueden usar los clientes para interactuar con el servicio. El elemento de servicio tam-

bién puede incluir cualquier número de *atributos*, que pueden ser cualquier objeto. Algunos atributos potenciales son iconos, clases que proporcionan IGU para el servicio, y objetos que proporcionan más información del servicio.

Los objetos de servicio suelen implementar uno o más interfaces a través de los cuales los clientes interactúan con el servicio. Por ejemplo, un servicio de búsqueda es un servicio Jini, y su objeto de servicio está en el registrador de servicios. El método **register( )** invocado por los proveedores de servicio durante el join se declara en la interfaz **ServiceRegistrar** (miembro del paquete **net.jini.core.lookup**), que implementan todos los objetos registradores de servicios. Los clientes y proveedores de servicios se comunican con el servicio de búsqueda a través del objeto registrador de servicios invocando a los métodos declarados en la interfaz **ServiceRegistrar**. De manera análoga, una unidad de disco proporcionaría un objeto servicio que implementaba alguna interfaz de servicio de almacenamiento bien conocido. Los clientes buscarían e interactuarían con la unidad de disco a través de esta interfaz de servicio de almacenamiento.

El proceso lookup

Una vez que se ha registrado un servicio con un servicio lookup vía el proceso join, ese servicio está disponible para ser usado por clientes que pregunten al servicio de búsqueda. Para construir un sistema distribuido que trabaje en conjunción para desempeñar alguna tarea un cliente debe localizar y agrupar la ayuda de servicios individuales. Para encontrar un servicio, los clientes preguntan a servicios de búsqueda vía un proceso llamado *lookup*.

Para llevar a cabo una búsqueda, un cliente invoca al método **lookup( )** de un objeto registrador de servicios. (Un cliente, como un proveedor de servicios, logra un registrador de servicios a través del proceso de *discovery* anteriormente descrito). El cliente pasa una plantilla de servicio como parámetro al **lookup( )**. Esta plantilla es un objeto que sirve como criterio de búsqueda para la consulta. La plantilla puede incluir una referencia a un array de objetos **Class**. Estos objetos **Class** indican al servicio de búsqueda el tipo (o tipos) Java del objeto servicio deseados por el cliente. La plantilla también puede incluir un *ID de servicio*, que identifica un servicio de manera unívoca, y atributos, que deben casar exactamente con los atributos proporcionados por el proveedor de servicios en el elemento de servicio. La plantilla de servicio también puede contener comodines para alguno de estos campos. Un comodín en el campo ID del servicio, por ejemplo, casará con cualquier ID de servicio. El método **lookup( )** envía la plantilla al servicio *lookup*, que lleva a cabo la consulta y devuelve cero a cualquier objeto de servicio que case. El cliente logra una referencia a los objetos de servicio que casen como valor de retorno del método **lookup( )**.

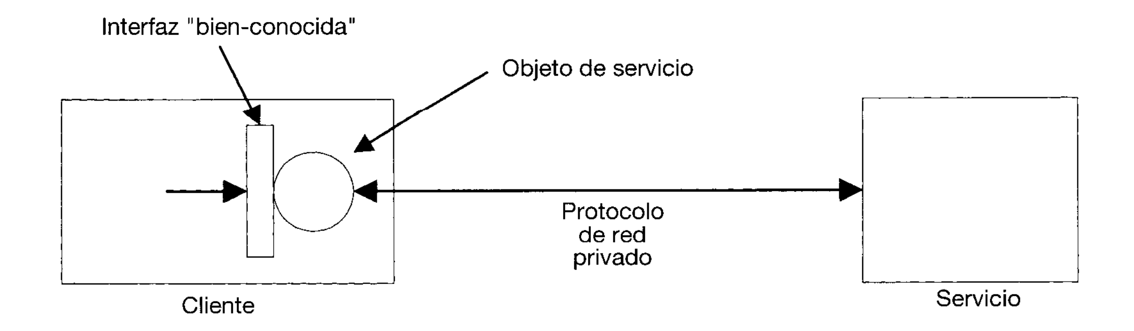
En el caso general, un cliente busca un servicio por medio de un tipo Java, generalmente una interfaz. Por ejemplo, si un cliente quisiese usar una impresora, compondría una plantilla de servicio que incluiría un objeto **Class** para una interfaz bien conocida. El servicio de búsqueda devolvería un objeto (o varios) de servicio que implementa esta interfaz. En la plantilla de servicio pueden incluirse atributos para reducir el número de coincidencias para una búsqueda de este tipo. El cliente usaría el servicio de impresión invocando a los métodos de la interfaz de impresión bien conocida del objeto de servicio.

Separación de interfaz e implementación

La arquitectura de Jini acerca la programación orientada a la red permitiendo a los servicios de red tomar ventaja de uno de los aspectos fundamentales de los objetos: la separación de interfaz e implementación. Por ejemplo, un objeto de servicio puede garantizar a los clientes acceso al servicio de muchas formas. El objeto puede representar, de hecho, todo el servicio, que es descargado al cliente durante la búsqueda, y después ejecutado localmente. Después, cuando el cliente invoca a métodos del objeto servicio, envía las peticiones al servidor a través de la red, que hace el trabajo real. Una tercera opción es que el objeto servicio local y un servidor remoto hagan parte del trabajo cada uno.

Una consecuencia importante de la arquitectura de Jini es que el protocolo de red usado para comunicarse entre un objeto de servicio intermediario y un servidor remoto no tiene por qué ser conocido por el cliente. Como se muestra en la siguiente figura, el protocolo de red es parte de la implementación del servicio. El cliente puede comunicarse con el servicio vía el protocolo privado porque el servicio inyecta parte de su propio código (el objeto servicio) en el espacio de reacciones del cliente. El objeto de servicio inyectado podría comunicarse con el servicio vía RMI, CORBA, DCOM, algún protocolo casero construido sobre sockets y flujos, o cualquier otra cosa. El cliente simplemente no tiene que preocuparse de protocolos de red, puede comunicarse con la interfaz bien conocida que implementa el objeto de servicio. El objeto de servicio se encarga de cualquier comunicación necesaria por la red.

Distintas implementaciones de una misma interfaz de servicio pueden usar enfoques y protocolos de red completamente diferentes. Un servicio puede usar hardware especializado para completar las solicitudes del cliente, o puede hacer todo su trabajo vía software. De hecho, el enfoque de implementación tomado por un único servicio puede evolucionar con el tiempo. El cliente puede estar seguro de tener un objeto de servicio que entiende la implementación actual del servicio, porque el cliente recibe el objeto de servicio (por parte del servicio de búsqueda) del propio proveedor de servicios. Para el cliente, un servicio tiene el aspecto de una interfaz bien conocida, independientemente de cómo esté implementado el servicio.



El cliente se comunica con el servicio a través de una interfaz bien conocida.

Abstraer sistemas distribuidos

Jini intenta elevar el nivel de abstracción para programación de sistemas distribuidos, desde el nivel del protocolo de red al nivel de interfaz del objeto. En la proliferación emergente de dispositivos embebidos conectados a redes puede haber muchas piezas de un sistema distribuido que provengan de distintos fabricantes. Jini hace innecesario que los fabricantes de dispositivos se pongan de acuerdo en los protocolos de nivel de red que permitan a sus dispositivos interactuar. En vez de ello, los fabricantes deben estar de acuerdo en las interfaces Java a través de las cuales pueden interactuar sus dispositivos. Los procesos de *discovery, join*, y *lookup* proporcionados por la infraestructura de tiempo de ejecución de Jini permiten a los dispositivos localizarse entre sí a través de la red. Una vez que se localizan, los dispositivos pueden comunicarse mutuamente a través de interfaces Java.

Resumen

Junto con Jini para redes de dispositivos locales, este capítulo ha presentado algunos, pero no todos los componentes a los que Sun denomina J2EE: la *Java 2 Enterprise Edition*. La meta de J2EE es construir un conjunto de herramientas que permitan a un desarrollador Java construir aplicaciones basadas en servidores mucho más rápido, y de forma independiente de la plataforma. Construir aplicaciones así no sólo es difícil y consume tiempo, sino que es especialmente difícil construirlas de forma que puedan ser portadas fácilmente a otras plataformas, y además mantener la lógica de negocio separada de los detalles de implementación subyacentes. J2EE proporciona un marco de trabajo para ayudar a crear aplicaciones basadas en servidores; estas aplicaciones tienen gran demanda hoy en día, y esa demanda parece, además, creciente.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Compilar y ejecutar los programas **ServidorParlante** y **ClienteParlante** de este capítulo. Ahora, editar los archivos para eliminar todo *espacio de almacenamiento intermedio* en las entradas y salidas, después compilar y volver a ejecutarlos para observar los resultados.

2. Crear un servidor que pida una contraseña, después un archivo y enviarlo a través de la conexión de red. Crear un cliente que se conecte al servidor, proporcionar la contraseña adecuada, y después capturar y salvar el archivo. Probar los dos programas en la misma máquina usando el **localhost** (la dirección IP de bucle local **127.0.0.1** producida al invocar a **InetAddress.getByName(null)**).

3. Modificar el servidor del Ejercicio 2, de forma que use el multihilo para manejar múltiples clientes.

4. Modificar **ClienteParlante.java** de forma que no se dé el vaciado de la salida y se observe su efecto.

5. Modificar **ServidorMultiParlante** de forma que use *hilos cooperativos*. En vez de lanzar un hilo cada vez que un cliente se desconecte, el hilo debería pasar a un "espacio de hilos disponibles". Cuando se desee conectar un nuevo cliente, el servidor buscará en este espacio un hilo que gestione la petición, y si no hay ninguno disponible, construirá uno nuevo. De esta forma, el número de hilos necesario irá disminuyendo en cuanto a la cantidad necesaria. La aportación de hilos cooperativos es que no precisa de sobrecarga para la creación y destrucción de hilos nuevos por cada cliente.

6. A partir de **MostrarHTML.java**, crear un *applet* que sea una pasarela protegida por contraseña a una porción particular de un sitio web.

7. Modificar **CrearTablasCID.java,** de forma que lea las cadenas de caracteres SQL de un texto en vez de **CIDSQL**.

8. Configurar el sistema para que se pueda ejecutar con éxito **CrearTablasCID.java** y **CargarBD.java**.

9. Modificar **ReglasServlets.java** superponiendo el método **destroy( )** para que salve el valor de **i** a un archivo, y el método **init( )** para que restaure el valor. Demostrar que funciona volviendo a arrancar el contenedor de servlets. Si no se tiene un contenedor de servlets, habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

10. Crear un servlet que añada un *cookie* al objeto respuesta, almacenándolo en el lado cliente. Añadir al servlet el código que recupera y muestra el *cookie*. Si no se tiene un contenedor de servlets habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache,org* para poder ejecutar servlets.

11. Crear un servlet que use un objeto **Session** para almacenar la información de sesión que se seleccione. En el mismo servlet, retirar y mostrar la información de sesión. Si no se tiene un contenedor de servlets habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

12. Crear un servlet que cambie el intervalo inactivo de una sesión a 5 segundos invocando a **getMaxInactiveInterval( )**. Probar que la función llegue a expirar tras 5 segundos. Si no se tiene un contenedor de servlets habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

13. Crear una página JSP que imprima una línea de texto usando la etiqueta <H1>. Poner el color de este texto al azar, utilizando código Java embebido en la página JSP. Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

14. Modificar el valor de la edad máxima de **Cookies.jsp** y observar el comportamiento bajo dos navegadores diferentes. Notar también la diferencia entre revisitar la página y apagar y volver a arrancar el navegador. Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

15. Crear un JSP con un campo que permita al usuario introducir el tiempo de expiración de la sesión y un segundo campo que guarde la fecha almacenada en la sesión. El botón de envío refresca la página y captura la hora de expiración actual además de la información de la sesión y las coloca como valores por defecto en los campos antes mencionados. Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

16. (Más complicado) tomar el programa **BuscarV.java** y modificarlo, de forma que al hacer clic en el nombre resultante tome automáticamente el nombre y lo copie al portapapeles (de forma que se pueda simplemente pegar en el correo electrónico). Habrá que echar un vistazo al Capítulo 13 para recordar como usar el portapapeles en JFC.

A: Paso y Retorno de Objetos

Hasta este momento el lector debería sentirse razonablemente cómodo con la idea de que cuando se está "pasando" un objeto se está pasando de hecho una referencia.

En muchos lenguajes de programación se puede usar la forma "normal" de pasar objetos, y la mayoría de veces funciona bien. Pero siempre parece que llega un momento en el que hay que hacer algo que se sale de la norma, y de repente todo se vuelve algo más complicado (o, en el caso de C++, bastante complicado). Java no es una excepción, y es importante que se entienda exactamente lo que ocurre al pasar y manipular objetos. Este apéndice pretende dar esta visión.

Otra forma de afrontar la cuestión de este apéndice, si se proviene de un lenguaje de programación bien equipado es, "¿tiene Java punteros?". Hay quien dice que los punteros son difíciles y peligrosos, y por consiguiente, malos, y dado que Java es todo virtud y bondad, tanto que te llevará al paraíso de la programación, no puede tener de esas cosas tan malas. Sin embargo, es más exacto decir que Java tiene punteros; de hecho, todo identificador en Java (excepto en el caso de los datos primitivos), es uno de esos punteros, pero su uso está restringido y reservado no sólo al compilador sino también al sistema de tiempo de ejecución. Hasta la fecha, yo les he llamado "referencias", y se puede pensar que son "punteros de seguridad", no muy distintos de las tijeras con punta roma de pre-escolar —no están afiladas, y por tanto no te puedes cortar sencillamente, pero pueden ser en ocasiones lentas y tediosas.

Pasando referencias

Cuando se pasa una referencia a un método, se sigue apuntando al mismo objeto. Un experimento simple puede demostrar esto:

```
//: apendicea:PasarReferencias.java
// Pasando referencias.

public class PasarReferencias {
  static void f(PasarReferencias h) {
    System.out.println("h dentro de f(): " + h);
  }
  public static void main(String[] args) {
    PasarReferencias p = new PasarReferencias();
    System.out.println("p dentro de main(): " + p);
    f(p);
  }
```

```
} ///:~
```

En las sentencias de impresión se invoca automáticamente al método **toString( )**, y **PasarReferencias** hereda directamente de **Object** sin redefinir **toString( )**. Por consiguiente, se usa la versión de **toString( )** de **Object**, que imprime la clase del objeto seguida de la dirección donde está localizado el mismo (no la referencia, sino el almacenamiento de objetos en sí). La salida tiene este aspecto:

```
p dentro de main(): PasarReferencias@1653748
h dentro de f(): PasarReferencias@1653748
```

Como puede verse, tanto **p** como **h** hacen referencia al mismo objeto. Esto es mucho más eficiente que duplicar un nuevo objeto **PasarReferencias** de forma que se pueda enviar un parámetro a un método. Pero trae un aspecto muy importante.

Uso de alias

El uso de alias significa que hay más de una referencia vinculada al mismo objeto, como en el ejemplo de arriba. El problema con el uso de alias radica en que alguien *escriba* en ese objeto. Si los propietarios de las referencias no esperan que el objeto varíe, se sorprenderán. Esto puede demostrarse con este sencillo ejemplo:

```
//: apendicea:Alias1.java
// Uso de alias: dos referencias al mismo objeto.

public class Alias1 {
  int i;
  Alias1(int ii) { i = ii; }
  public static void main(String[] args) {
    Alias1 x = new Alias1(7);
    Alias1 y = x; // Asignar la referencia
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
    System.out.println("Incrementando x");
    x.i++;
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
  }
} ///:~
```

En la línea:

```
Alias1 y = x; // Asignar la referencia
```

se crea una nueva referencia **Alias1**, pero en vez de ser asignada a un nuevo objeto creado con **new**, se asigna a una referencia existente. Por tanto, los contenidos de la referencia **x**, que es la dirección a la que apunta el objeto **x**, se asignan a **y**, con lo que tanto **x** como **y** están asignadas al mismo objeto. Por tanto, cuando en la sentencia:

```
x.i++;
```

se incrementa la **i** de **x**, también se verá afectada la **i** de **y**. Esto puede verse en la salida:

```
x: 7
y: 7
Incrementando x
x: 8
y: 8
```

Una buena solución en este caso es simplemente no hacerlo: no establecer conscientemente un alias, o más de una referencia a un objeto dentro de un mismo ámbito. El código será así más fácil de depurar y entender. Sin embargo, cuando se está pasando una referencia como un argumento —que es como Java se supone que funciona— se está generando automáticamente un alias porque la referencia local creada puede modificar el "objeto externo" (el objeto que se creó fuera del ámbito del método). He aquí un ejemplo:

```
//: apendicea:Alias2.java
// Las llamadas a metodos implican alias
// de sus parámetros.

public class Alias2 {
  int i;
  Alias2(int ii) { i = ii; }
  static void f(Alias2 referencia) {
    referencia.i++;
  }
  public static void main(String[] args) {
    Alias2 x = new Alias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Invocando a f(x)");
    f(x);
    System.out.println("x: " + x.i);
  }
} ///:~
```

La salida es:

```
x: 7
Invocando a f(x)
x: 8
```

El método modifica su argumento, el objeto externo. Cuando se da este tipo de situación, hay que decidir si tiene sentido, si el usuario lo espera, y si va a causar o no problemas.

En general, se invoca a un método para producir un valor de retorno y/o un cambio de estado en el objeto *por el que se llama al método*. (Un método es el cómo se "envía un mensaje" a ese objeto.) Es menos común llamar a un método para que manipule sus argumentos; a esto se le denomina "lla-

mar a un método por sus *efectos laterales"*. Por consiguiente, al crear un método que modifica sus parámetros, el usuario debe estar instruido y advertido sobre el uso de ese método y sus potenciales sorpresas. Debido a posibles confusiones y fallos, es mucho mejor evitar cambiar el parámetro.

Si hay que modificar un parámetro durante una llamada a un método y no se pretende modificar el parámetro externo, entonces se debería proteger ese parámetro haciendo una copia dentro del método. Sobre esto versará gran parte de este apéndice.

Haciendo copias locales

Para repasar: todo paso de parámetros en Java se lleva a cabo pasando referencias. Es decir, al pasar "un objeto", verdaderamente sólo se está pasando una referencia a un objeto que reside fuera del método, por lo que si se llevan a cabo modificaciones con esa referencia, se modifica el objeto externo. Además:

- Durante el paso de parámetros se produce automáticamente el uso de alias.
- No hay objetos locales, sólo referencias locales.
- Las referencias tienen ámbitos, los objetos no.
- La longevidad de los objetos nunca es un problema en Java.
- No hay soporte por parte del lenguaje (por ejemplo, la palabra clave "const") para evitar que se modifiquen los objetos (es decir, para evitar los aspectos negativos del uso de alias).

Si sólo se está leyendo información de un objeto sin modificarlo, el paso de una referencia es la forma más eficiente de paso de parámetros. Está bien que la forma de hacer las cosas por defecto sea la más eficiente. Sin embargo, en ocasiones es necesario ser capaz de tratar el objeto como si fuera "local", de forma que los cambios que se hagan sólo afecten a la copia local, sin modificar el objeto externo. Muchos lenguajes de programación soportan la habilidad de hacer automáticamente una copia local del objeto externo, dentro del método[1]. Java no lo hace, pero te permite producir este efecto.

Paso por valor

Así surge el aspecto de la terminología, que siempre suele ser adecuado. El término "pasar por valor" y su significado dependen de cómo se perciba el funcionamiento del programa. El significado general es que se logra una copia de sea lo que sea lo que se pasa, pero la pregunta real es cómo se piensa en lo que se pasa. Cuando se "pasa por valor", hay dos visiones claramente distintas:

---

[1] En C, que generalmente manipula pequeñas cantidades de datos, el paso por defecto es por valor. C++ tenía que seguir este esquema, pero con objetos, el *paso por valor* no suele ser la forma más eficiente. Además, la codificación de clases para dar soporte en C++ al paso por valor, no supone sino quebraderos de cabeza.

1. Java pasa todo por valor. Cuando se pasan datos primitivos a un método, se logra una copia aparte del dato. Cuando se pasa una referencia a un método, se obtiene una referencia al método, se puede lograr una copia de la referencia. Ergo, todo se pasa por valor. Por supuesto, se supone que siempre se piensa (y se tiene cuidado en qué) que se están pasando referencias, pero parece que el diseño de Java ha ido mucho más allá, permitiéndote ignorar (la mayoría de las veces) que se está trabajando con una referencia. Es decir, parece permitirte pensar en la referencia como si se tratara "del objeto" puesto que implícitamente se desreferencia cuando se hace una llamada a un método.

2. Java pasa los tipos primitivos de datos por valor (sin que haya parámetros), pero los objetos se pasan por referencia. Ésta es la visión de que la referencia es un alias del objeto, por lo que *no* se piensa en el paso de referencias, sino que se dice "Estoy pasando el objeto". Dado que no se logra una copia local del objeto, cuando se pasa a un método, claramente, los objetos no se pasan por valor. Parece haber cierto soporte para esta visión por parte de Sun, pues una de las palabras clave "no implementada pero reservada" era **byvalue**. (No se sabe, sin embargo, si esa palabra clave algún día llegará a ver la luz).

Habiendo visto ambas perspectivas, y tras decir que "depende de cómo vea cada uno lo que es una referencia", intentaré dejar este aspecto de lado. Al final, no es *tan* importante —lo que es importante es que se entienda que pasar una referencia permite que el objeto que hizo la llamada pueda cambiar de forma inesperada.

Clonando objetos

La razón más probable para hacer una copia local de un objeto es cuando se va a modificar este objeto y no se desea modificar el objeto llamador. Si se decide que se desea hacer una copia local, basta con usar el método **clone( )** para llevar a cabo la operación. Se trata de un método definido como **protected** en la clase base **Object**, y que hay que superponer como **public** en cualquier clase derivada que se desee clonar. Por ejemplo, la clase de biblioteca estándar **ArrayList** superpone **clone( )**, por lo que se puede llamar a **clone( )** para **ArrayList**:

```
//: apendicea:Clonar.java
// La operación clone() funciona sólo para unos pocos
// elementos en la biblioteca estándar de Java.
import java.util.*;

class Int {
  private int i;
  public Int(int ii) { i = ii; }
  public void incrementar() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
}
```

```
public class Clonar {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Incrementar todos los elementos de v2:
    for(Iterator e = v2.iterator();
        e.hasNext(); )
      ((Int)e.next()).incrementar();
    // Ver si los elementos de v han cambiado:
    System.out.println("v: " + v);
  }
} ///:~
```

El método **clone( )** produce un **Object**, que hay que convertir al método apropiado. El ejemplo muestra cómo el método **clone( )** de **ArrayList** *no* intenta clonar automáticamente todos los objetos contenidos en el **ArrayList** —el viejo **ArrayList** y el **ArrayList** clonado son alias del mismo objeto. A esto se le suele llamar hacer una *copia superficial*, puesto que sólo se copia la porción de "superficie" del objeto. El objeto en sí consiste en esta "superficie" más todos los objetos a los que apunten las referencias, más todos los objetos a los que *esos* objetos apuntan, etc. A esto se le suele llamar la "telaraña de objetos". Copiar absolutamente todo se llama hacer una *copia en profundidad.*

El efecto de la copia superficial puede verse en la salida, donde las acciones hechas sobre **v2** afectan a **v**:

```
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Ahora, intentar **clone( )** (clonar) los objetos del **ArrayList** es probablemente una simple presunción, puesto que no hay ninguna garantía de que estos objetos sean "clonables"[2].

Añadiendo a una clase la capacidad de ser clonable

Incluso aunque el método para clonar está definido en la clase **Object**, base de todas las clases, clonar *no* es algo que esté disponible automáticamente para todas las clases[3]. Esto parecería contra-intuitivo con la idea de que los métodos de la clase base siempre están disponibles para sus clases derivadas. En Java, clonar va contra esta idea; si se desea que exista para una clase, hay que añadir código de forma específica, para que la clonación sea posible.

---

[2] Ésta no es la palabra conforme aparece en el diccionario, pero es la usada en la librería Java, así que es la usada aquí con la esperanza de reducir la confusión.

[3] Aparentemente, se puede crear un ejemplo simple para esta sentencia, así (ver continuación de la nota en la página siguiente):

Usando un truco con **protected**

Para evitar tener la capacidad de clonar por defecto toda clase que se cree, el método **clone( )** es **protected** y pertenece a la clase base **Object**. Esto no sólo significa que no está disponible por defecto para el programador cliente que simplemente use la clase (aquél que no genera subclases de las mismas), sino que también significa que no se puede llamar a **clone( )** vía referencia a la clase base. (Aunque eso podría parecer útil en algunas situaciones, como pudiera ser clonar de forma polimórfica un conjunto de **Objects**.) Es, en efecto, una forma de proporcionar al programador, en tiempo de compilación, la información de que el objeto no es clonable —y de hecho, la mayoría de las clases de la biblioteca estándar de Java no son clonables. Por tanto, si se dice:

```
Integer x = new Integer(1);
x = x.clone();
```

En tiempo de compilación se obtendrá un mensaje de error que indica que **clone( )** no es accesible (puesto que **Integer** no lo superpone y por defecto se acude a la versión **protected**).

Si, sin embargo, se está en una clase derivada de **Object** (cosa que son todas las clases), se puede invocar a **Object.clone( )** puesto que es **protected** y la clase es una descendiente. La clase base **clone( )** tiene funcionalidad útil —lleva a cabo la duplicación del *objeto de la clase derivada*, actuando por consiguiente como la operación común de clonado. Sin embargo, hay que hacer **public** la operación de clonado de *cada uno* si se desea que ésta esté accesible. Por tanto, dos aspectos vitales al clonar son:

- Llamar casi siempre a **super.clone( )**
- Hacer **public** la función **clone** de cada uno.

Probablemente, se deseará superponer **clone( )** en subsiguientes clases derivadas, pues de otra forma se usará el nuevo (y ahora **public**) método **clone( )**, y eso podría no ser siempre lo correcto (aunque, puesto que **Object.clone( )** hace una copia del objeto, puede que sí). El truco **protected** sólo funciona una vez —la primera vez que se hereda de una clase que no es clonable y se desea hacer que sí lo sea. En cualquier clase heredada de la ya definida, estará disponible el método **clone( )** puesto que Java no puede reducir el acceso a métodos durante la derivación. Es decir, una

---

```
public class Cloneit implements Cloneable {
  public static void main (String [] args)
  throws CloneNotSupporteException {
    Cloneit a = new Cloneit();
    Cloneit b = (Cloneit)a.clone();
  }
}
```

Sin embargo, esto sólo funciona porque **main( )** es un método de **Cloneit** y, por consiguiente, tiene permiso para llamar al método **protected clone( )** de la clase base. Si se le llama desde una clase diferente, no compilará.

vez que una clase es clonable, todo lo que se derive de la misma también lo es, a no ser que se proporcionen mecanismos (descritos más adelante) para "desactivar" la "clonabilidad".

Implementando la interfaz Cloneable

Todavía se necesita algo más para completar la "clonabilidad" de un objeto: implementar la interfaz **Cloneable**. Esta **interfaz** es un poco extraña pues ¡está vacía!

```
interface Cloneable()
```

La razón de implementar esta interfaz vacía no es obviamente porque se vaya a aplicar un molde hacia arriba a **Cloneable** e invocar a uno de sus métodos. Aquí, el uso de la interfaz se considera como una especie de "intrusismo" pues usa una faceta para algo que no es su propósito original. Implementar la interfaz **Cloneable** actúa como indicador, vinculado al tipo de la clase.

Hay dos razones para la existencia de la interfaz **Cloneable**. En primer lugar, se podría tener una referencia convertida al tipo base sin saber si es posible hacer una clonación a ese objeto. En este caso, se puede usar la palabra clave **instanceof** (descrita en el Capítulo 12) para averiguar si la referencia está conectada a un objeto que puede clonarse:

```
if (miReferencia instanceof Cloneable) // ...
```

La segunda razón es que está en este diseño porque se pensaba que la "clonabilidad" probablemente no fuera deseable para todos los tipos de objetos. Por tanto, **Object.clone( )** verifica que una clase implemente la interfaz **Cloneable**. Si no, lanza una excepción **CloneNotSupportedException**. Por tanto, en general, uno se ve obligado a implementar **Cloneable** como parte del soporte a la clonación.

Clonación con éxito

Una vez comprendidos los detalles de implementación del método **clone( )**, ya se pueden crear clases que pueden duplicarse sencillamente para proporcionar una copia local:

```
//: apendicea:CopiaLocal.java
// Creando copias locales con clone().
import java.util.*;

class MiObjeto implements Cloneable {
  int i;
  MiObjeto(int ii) { i = ii; }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("MiObjeto no es clonable");
    }
```

```
    return o;
  }
  public String toString() {
    return Integer.toString(i);
  }
}

public class CopiaLocal {
  static MiObjeto g(MiObjeto v) {
    // El paso de una referencia modifica el objeto externo:
    v.i++;
    return v;
  }
  static MiObjeto f(MiObjeto v) {
    v = (MiObjeto)v.clone(); // Copia local
    v.i++;
    return v;
  }
  public static void main(String[] args) {
    MiObjeto a = new MiObjeto(11);
    MiObjeto b = g(a);
    // Probando la equivalencia de referencias, ,
    // no la equivalencia de objetos:
    if(a == b)
      System.out.println("a == b");
    else
      System.out.println("a != b");
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    MiObjeto c = new MiObjeto(47);
    MiObjeto d = f(c);
    if(c == d)
      System.out.println("c == d");
    else
      System.out.println("c != d");
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
} ///:~
```

En primer lugar, **clone( )** debe ser accesible, por lo que hay que hacerlo **public**. En segundo lugar, para la parte inicial de la operación **clone( )** habría que invocar a la versión de **clone( )** de la clase base. El **clone( )** al que se está invocando aquí es el predefinido dentro de **Object**, y se puede invocar por ser **protected**, y por tanto, accesible en las clases derivadas.

El método **Object.clone( )** averigua lo grande que es el objeto, crea memoria suficiente para uno nuevo, y copia todos los bits del viejo al nuevo. A esto se le llama *copia bit a bit*, y es lo que generalmente se esperaría que hiciera un método **clone( )**. Pero antes de que **Object.clone( )** lleve a cabo sus operaciones, primero comprueba si una clase es **Cloneable** —es decir, si implementa o no la interfaz **Cloneable**. Si no lo hace, **Object.clone( )** lanza una **CloneNotSupportedException** para indicar que no se puede clonar. Por tanto, hay que llevar la llamada a **super.clone( )** con un bloque *try-catch*, que capture una excepción que nunca debería darse (por haber implementado la interfaz **Cloneable**).

En **CopiaLocal**, los dos métodos **g( )** y **f( )** demuestran la diferencia entre los dos enfoques en el paso de parámetros. El método **g( )** muestra el paso por referencia en el que se modifica el objeto exterior, devolviendo una referencia a ese objeto exterior, mientras que **f( )** clona el parámetro, desacoplándolo y dejando a salvo el objeto original. Después, puede hacer lo que desee, e incluso devolver una referencia al nuevo objeto sin crear efectos perniciosos al original. Nótese la sentencia, en cierta medida curiosa:

```
v = (MiObjeto)v.clone();
```

Es aquí donde se crea la copia local. Para evitar confusiones por sentencias así, recuérdese que este dialecto de codificación tan extraño está totalmente permitido en Java, puesto que todo identificador de objeto es de hecho una referencia al mismo. Por tanto, se usa la referencia a **v** para **clone( )** una copia de aquello a lo que hace referencia, y éste devuelve una referencia al tipo base **Object** (porque así está definido en **Object.clone( )**) que hay que convertir después al tipo adecuado.

En **main( )**, se prueba la diferencia entre los efectos de los dos enfoques de paso de parámetros en los dos métodos. La salida es:

```
a == b
a = 12
b = 12
c != d
c = 47
d = 48
```

Es importante darse cuenta de que las pruebas de equivalencia en Java no comparan la parte interna de los objetos para ver si sus valores son el mismo. Los operadores == y != simplemente comparan las *referencias*. Si las direcciones contenidas en las referencias son iguales, entonces apuntan al mismo objeto siendo por tanto "iguales". Por tanto ¡lo que verdaderamente prueban los operadores es si las referencias son alias de un mismo objeto!

El efecto de **Object.clone( )**

¿Qué es lo que de verdad ocurre cuando se invoca a **Object.clone( )** que hace tan esencial llamar a **super.clone( )** cuando se superpone **clone( )** en una clase? El método **clone( )** de la clase raíz es el responsable de crear la cantidad de almacenamiento correcta y de hacer la copia bit a bit del objeto original al espacio de almacenamiento del nuevo objeto. Es decir, no simplemente crea el es-

pacio de almacenamiento y se encarga de la copia de un **Object** —de hecho averigua el tamaño exacto del objeto que está copiando y lo duplica. Dado que todo esto ocurre a partir del código del método **clone( )** definido en la clase raíz (que no tiene idea de qué es lo que se ha heredado de la misma), como puede adivinarse, el proceso implica RTTI para determinar el objeto en concreto que está siendo clonado, y hacer una copia de bits correcta para ese tipo.

Sea lo que sea lo que se haga, la primera parte del proceso de clonado debería ser normalmente una llamada a **super.clone( )**. Así se echan las bases de la operación de clonado creando un duplicado exacto. En este momento, se pueden llevar a cabo otras operaciones necesarias para completar el clonado.

Para asegurarse de cuáles son esas operaciones, hay que entender exactamente qué es lo que hace exactamente **Object.clone( )**. En concreto ¿clona automáticamente el destino de todas las referencias? El ejemplo siguiente permite probar la respuesta a esta pregunta:

```
//: apendicea:Serpiente.java
// Probar el clonado para ver si también
// se clona el destino de las referencias.

public class Serpiente implements Cloneable {
  private Serpiente siguiente;
  private char c;
  // Valor de I == número de segmentos
  Serpiente(int i, char x) {
    c = x;
    if(--i > 0)
      siguiente = new Serpiente(i, (char)(x + 1));
  }
  void incrementar() {
    c++;
    if(siguiente != null)
      siguiente.incrementar();
  }
  public String toString() {
    String s = ":" + c;
    if(siguiente != null)
      s += siguiente.toString();
    return s;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Serpiente no se puede clonar");
    }
```

```
      return o;
   }
   public static void main(String[] args) {
      Serpiente s = new Serpiente(5, 'a');
      System.out.println("s = " + s);
      Serpiente s2 = (Serpiente)s.clone();
      System.out.println("s2 = " + s2);
      s.incrementar();
      System.out.println(
         "tras s.incrementar, s2 = " + s2);
   }
} ///:~
```

Una **Serpiente** está compuesta por un conjunto de segmentos, cada uno de tipo **Serpiente**. Por consiguiente, es una lista simplemente enlazada. Los segmentos se crean de forma recursiva, decreciendo el primer parámetro al constructor en cada segmento hasta llegar a cero. Para dar a cada segmento una etiqueta única, se incrementa el segundo parámetro, un **char**, por cada llamada recursiva al constructor.

El método **incrementar( )** incrementa cada etiqueta de forma que pueda verse el cambio, y el **toString( )** imprime cada etiqueta de forma recursiva. La salida es:

```
s = :a:b:c:d:e
s2 = :a:b:c:d:e
tras s.incrementar, s2 = a:c:d:e:f
```

Esto significa que **Object.clone( )** sólo duplica el primer segmento, y por consiguiente, hace una copia superficial. Si se desea duplicar toda la serpiente —una copia en profundidad— hay que llevar a cabo las operaciones adicionales dentro del **clone( )** superpuesto.

Generalmente, se invocará a **super.clone( )** en cualquier clase derivada de una clase clonable para asegurarse de que se den todas las operaciones de la clase base (incluida **Object.clone( )**). A continuación se hace una llamada explícita a **clone( )** por cada referencia que haya en el objeto; de otra forma, estas referencias se convertirían en alias de las del objeto original. Es análogo a la forma de llamar a los constructores —primero el constructor de la clase base, después el constructor de la siguiente derivada, y así hasta el constructor de la última clase derivada. La diferencia reside en que **clone( )** no es un constructor, por lo que no hay nada que haga automáticamente. Hay que asegurarse de hacerlo a mano.

Clonando un objeto compuesto

Al intentar hacer una copia en profundidad de un objeto compuesto, hay un problema. Hay que asumir que el método **clone( )** de los objetos miembros de hecho llevará a cabo una copia en profundidad de *sus* referencias, y así sucesivamente. Esto es bastante comprometido. Significa de hecho que para que funcione una copia en profundidad, uno debe o controlar todo el código en todas sus clases, o al menos tener el conocimiento suficiente sobre todas las clases involucradas en la copia

en profundidad como para saber que están llevando a cabo su copia en profundidad de forma correcta.

Este ejemplo muestra qué es lo que hay que hacer para lograr una copia en profundidad cuando se manipule un objeto compuesto:

```
//: apendicea:CopiaProfundidad.java
// Clonando un objeto compuesto.

class LeerProfundidad implements Cloneable {
  private double Profundidad;
  public LeerProfundidad(double profundidad) {
    this.profundidad = profundidad;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    return o;
  }
}

class LeerTemperatura implements Cloneable {
  private long tiempo;
  private double temperatura;
  public LeerTemperatura(double temperatura) {
    tiempo = System.currentTimeMillis();
    this.temperatura = temperatura;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    return o;
  }
}

class LeerOceano implements Cloneable {
  private LeerProfundidad profundidad;
```

```
   private LeerTemperatura temperatura;
   public LeerOceano(double datost, double datosp){
      temperatura = new LeerTemperatura(datost);
      profundidad = new LeerProfundidad(datosp);
   }
   public Object clone() {
      LeerOceano o = null;
      try {
         o = (LeerOceano)super.clone();
      } catch(CloneNotSupportedException e) {
         e.printStackTrace(System.err);
      }
      // Hay que clonar las referencias:
      o.profundidad = (LeerProfundidad)o.profundidad.clone();
      o.temperatura =
         (LeerTemperatura)o.temperatura.clone();
      return o; // Conversión de Nuevo a Object
   }
}

public class CopiaProfundidad {
   public static void main(String[] args) {
      LeerOceano leer =
         new LeerOceano(33.9, 100.5);
      // Ahora clonarlo:
      LeerOceano l =
         (LeerOceano)leer.clone();
   }
} ///:~
```

LeerProfundidad y **LeerTemperatura** son bastante similares; ambos contienen sólo tipos de datos primitivos. Por consiguiente, el método **clone( )** puede ser bastante simple: llama a **super.clone( )** y devuelve el resultado. Nótese que en ambos casos el código de **clone( )** es idéntico.

LeerOceano está compuesto de objetos **LeerTemperatura** y **LeerProfundidad**, y así, para hacer una copia en profundidad, su **clone( )** debe clonar las referencias incluidas en **LeerOceano**. Para lograr esto, hay que convertir el resultado de **super.clone( )** a un objeto **LeerOceano** (de forma que se pueda acceder a las referencias **profundidad** y **temperatura**).

Una copia en profundidad con ArrayList

Revisitemos el ejemplo de **ArrayList** visto anteriormente en este apéndice. Esta vez, la clase **Int2** es clonable, por lo que se puede hacer una copia en profundidad del **ArrayList**:

```
//: apendicea:AniadirClonado.java
// Hay que dar unas pocas vueltas
```

```
// para añadir el clonado a una clase propia.
import java.util.*;

class Int2 implements Cloneable {
  private int i;
  public Int2(int ii) { i = ii; }
  public void incrementar() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Int2 no se puede clonar");
    }
    return o;
  }
}

// Una vez que es clonable, la herencia no
// elimina la "clonabilidad":
class Int3 extends Int2 {
  private int j; // Duplicado automáticamente
  public Int3(int i) { super(i); }
}

public class AniadirClonado {
  public static void main(String[] args) {
    Int2 x = new Int2(10);
    Int2 x2 = (Int2)x.clone();
    x2.incrementar();
    System.out.println(
      "x = " + x + ", x2 = " + x2);
    // Todo lo heredado es también clonable:
    Int3 x3 = new Int3(7);
    x3 = (Int3)x3.clone();

    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int2(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Ahora, clonar cada elemento:
```

```
        for(int i = 0; i < v.size(); i++)
          v2.set(i, ((Int2)v2.get(i)).clone());
        // Incrementar todos los elementos de v2:
        for(Iterator e = v2.iterator();
            e.hasNext(); )
          ((Int2)e.next()).incrementar();
        // Ver si los elementos de v han cambiado:
        System.out.println("v: " + v);
        System.out.println("v2: " + v2);
    }
} ///:~
```

Int3 hereda de **Int2** y se añade un nuevo miembro primitivo **int j**. Podría pensarse que hay que superponer de nuevo **clone( )** para asegurarse de la copia de **j**, pero no es así. Cuando se invoca al **clone( )** de **Int2** como el **clone( )** de **Int3**, llama a **Object.clone( )**, que determina que está trabajando con un **Int3** y duplica todos los bits del **Int3**. En la medida en que no se añaden referencias que necesiten ser clonadas, la llamada a **Object.clone( )** lleva a cabo toda la duplicación necesaria, independientemente de lo lejos que esté definido **clone( )** dentro de la jerarquía.

Aquí puede deducirse qué es necesario para hacer una copia en profundidad de un **ArrayList**: una vez clonado el **ArrayList**, hay que recorrer cada uno de los objetos apuntados por el **ArrayList**. Habría que hacer algo similar a esto si se desea hacer una copia en profundidad de un **HashMap**.

El resto del ejemplo muestra que se dio el clonado, mostrando que, una vez clonado un objeto, es posible modificarlo sin que el original se vea alterado.

Copia en profundidad vía serialización

Cuando se considera la serialización de objetos de Java (presentada en el Capítulo 11) podría observarse que si un objeto se serializa y después se deserializa, de hecho, está siendo clonado.

Por tanto ¿por qué no usar la serialización para llevar a cabo copias en profundidad? He aquí un ejemplo que comprueba los dos enfoques, cronometrándolos:

```
//: apendicea:Competir.java
import java.io.*;

class Cosa1 implements Serializable {}
class Cosa2 implements Serializable {
  Cosa1 o1 = new Cosa1();
}

class Cosa3 implements Cloneable {
  public Object clone() {
    Object o = null;
    try {
```

```
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Cosa3 no se puede clonar");
    }
    return o;
  }
}

class Cosa4 implements Cloneable {
  Cosa3 o3 = new Cosa3();
  public Object clone() {
    Cosa4 o = null;
    try {
      o = (Cosa4)super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Cosa4 no se puede clonar");
    }
    // Clonar también el campo:
    o.o3 = (Cosa3)o3.clone();
    return o;
  }
}

public class Competir {
  static final int TAMANIO = 5000;
  public static void main(String[] args)
  throws Exception {
    Cosa2[] a = new Cosa2[TAMANIO];
    for(int i = 0; i < a.length; i++)
      a[i] = new Cosa2();
    Cosa4[] b = new Cosa4[TAMANIO];
    for(int i = 0; i < b.length; i++)
      b[i] = new Cosa4();
    long t1 = System.currentTimeMillis();
    ByteArrayOutputStream buf =
      new ByteArrayOutputStream();
    ObjectOutputStream salida =
      new ObjectOutputStream(buf);
    for(int i = 0; i < a.length; i++)
      salida.writeObject(a[i]);
    // Ahora, hacerse con las copias:
    ObjectInputStream entrada =
      new ObjectInputStream(
        new ByteArrayInputStream(
          buf.toByteArray()));
```

```
      Cosa2[] c = new Cosa2[TAMANIO];
      for(int i = 0; i < c.length; i++)
        c[i] = (Cosa2)entrada.readObject();
      long t2 = System.currentTimeMillis();
      System.out.println(
        "Duplicacion via serializacion: " +
        (t2 - t1) + " Milisegundos");
      // Ahora, intentar clonar:
      t1 = System.currentTimeMillis();
      Cosa4[] d = new Cosa4[TAMANIO];
      for(int i = 0; i < d.length; i++)
        d[i] = (Cosa4)b[i].clone();
      t2 = System.currentTimeMillis();
      System.out.println(
        "Duplicacion via clonado: " +
        (t2 - t1) + " Milisegundos");
  }
} ///:~
```

Cosa2 y **Cosa4** contienen objetos miembro, de forma que se copie algo en profundidad. Es interesante darse cuenta de que mientras que es fácil configurar las clases **Serializable**, hay que hacer mucho más trabajo para duplicarlas. El clonado implica mucho trabajo para configurar las clases, pero la duplicación de objetos es relativamente simple. Los resultados hablan por sí solos. He aquí los resultados de tres ejecuciones:

```
Duplicacion via serializacion: 940 Milisegundos
Duplicacion via clonado: 50 Milisegundos

Duplicacion via serializacion: 710 Milisegundos
Duplicacion via clonado: 60 Milisegundos

Duplicacion via serializacion: 770 Milisegundos
Duplicacion via clonado: 50 Milisegundos
```

A pesar de la diferencia de tiempo tan significativa entre la serialización y el clonado, también se verá que la técnica de serialización parece fluctuar más en cuanto a duración, mientras que el clonado parece ser más estable.

Añadiendo "clonabilidad" a lo largo de toda una jerarquía

Si se crea una clase nueva, su clase base por defecto es **Object**, y por tanto, no clonable (como se verá en la sección siguiente). Mientras no se añada explícitamente "clonabilidad", ésta no surgirá por sí sola. Pero se puede añadir en cualquier capa y todas sus descendientes serán clonables:

```
//: apendicea:GolpeHorror.java
// La Clonabilidad puede insertarse
// en cualquier nivel de la herencia.
import java.util.*;

class Persona {}
class Heroe extends Persona {}
class Cientifico extends Persona
    implements Cloneable {
  public Object clone() {
    try {
      return super.clone();
    } catch(CloneNotSupportedException e) {
      // Esto no debería ocurrir nunca:
      // ¡Ya es clonable!
      throw new InternalError();
    }
  }
}
class CientificoLoco extends Cientifico {}

public class GolpeHorror {
  public static void main(String[] args) {
    Persona p = new Persona();
    Heroe h = new Heroe();
    Cientifico s = new Cientifico();
    CientificoLoco m = new CientificoLoco();

    // p = (Persona)p.clone(); // Error de compilación
    // h = (Heroe)h.clone(); // Error de compilación
    s = (Cientifico)s.clone();
    m = (CientificoLoco)m.clone();
  }
} ///:~
```

Antes de añadir "clonabilidad", el compilador evitaba que clonases cosas. Cuando se añadió "clonabilidad" a **Cientifico**, tanto esta clase como todas sus descendientes se convirtieron en clonables.

¿Por qué un diseño tan extraño?

Si todo esto parece seguir un esquema extraño, es porque así lo es. Uno podría preguntarse por qué es así. ¿Qué hay detrás de un diseño así?

Originalmente, Java se diseñó como un lenguaje para controlar cajas hardware y, desde luego, no con Internet en mente. En un lenguaje de propósito general como éste tiene sentido que el progra-

mador pueda clonar cualquier objeto. Por ello, se ubicó **clone( )** en la clase raíz **Object**, *pero* era un método **public** de forma que siempre se pudiera clonar cualquier objeto. Éste parecía ser el enfoque más flexible, y después de todo ¿qué daño podría hacer?

Bien, cuando Java se empezó a contemplar como el último lenguaje de programación de Internet, las cosas cambiaron. De repente, hay aspectos de seguridad, y por supuesto, estos objetos están relacionados con el uso de objetos, y necesariamente no se desea que nadie sea capaz de clonar los objetos de seguridad. Por tanto, lo que se ven son un montón de parches aplicados al esquema original, simple y directo: ahora **clone( )** es **protected** en **Object**. Hay que superponerlo *e* **implementar Cloneable** *y* hacer frente a las posibles excepciones.

Merece la pena reseñar que hay que usar la interfaz **Cloneable** *sólo* si se va a invocar al método **clone( )** de **Object**, puesto que este método comprueba en tiempo de ejecución si la clase implementa **Cloneable**. Pero por motivos de consistencia (y puesto que de cualquier forma, **Cloneable** está vacío) hay que implementarlo.

Controlando la "clonabilidad"

Uno podría sugerir que para eliminar la "clonabilidad" simplemente hace falta que se haga **private** el método **clone( )**, pero esto no funcionaría puesto que no se puede tomar un método de la clase base y hacerlo menos accesible en una clase derivada. Por tanto, no es tan simple. Y lo que es más, hay que poder controlar si se puede clonar el objeto. De hecho, en una clase que uno diseñe se pueden tomar varias actitudes a este respecto:

1. Indiferencia. No se hace nada con el clonado, lo que significa que tu clase no puede clonarse pero a una clase que se derive de la misma se le podría añadir clonación si se desea. Esto sólo funciona si el **Object.clone( )** por defecto hace algo razonable con todos los campos de la clase.

2. Soportar **clone( )**. Seguir la práctica estándar de implementar **Cloneable** y superponer **clone( )**. En el **clone( )** superpuesto se invoca a **super.clone( )** y se capturan todas las excepciones (de forma que el **clone( )** superpuesto no lance ninguna excepción).

3. Soportar el clonado condicionalmente. Si tu clase tiene referencias a otros objetos que podrían o no ser clonables (por ejemplo una clase contenedora) tu **clone( )** puede intentar clonar todos los objetos para los que se tenga referencias, y si lanzan excepciones, simplemente pasárselas al programador. Por ejemplo, considérese un tipo de **ArrayList** especial que intenta clonar todos los objetos que guarda. Cuando se escriba un **ArrayList** así, no se puede saber el tipo de objetos que el programador cliente podría poner en el **ArrayList**, por lo que no se sabe si pueden ser clonados.

4. No implementar **Cloneable** pero superponer **clone( )** como **protected**, produciendo el comportamiento de copia correcto para todos los campos. De esta forma, cualquiera que herede de esta clase puede superponer **clone( )** e invocar a **super.clone( )** para producir el comportamiento de copia correcto. Nótese que una implementación puede y debería invocar a **super.clone( )** incluso aunque ese método espere un objeto **Cloneable** (en caso contrario, lan-

zará una excepción), porque nadie la invocará directamente con un objeto del tipo creado. Sólo será invocada a través de clases derivadas, que, si se desea que funcione correctamente, deberán implementar **Cloneable**.

5. Intentar evitar el clonado no implementando **Cloneable** y superponiendo **clone( )** para que lance una excepción. Esto sólo tiene éxito si cualquier clase derivada de ésta llama a **super.clone( )** en su redefinición de **clone( )**. De otra forma, un programador podría volver a ello.

6. Evitar el clonado haciendo que la clase sea **final**. Hacer la clase **final** es la única forma de garantizar que se evite el clonado. Además, al tratar con objetos de seguridad o cualquier otra situación en la que se desee controlar el número de objetos que se crean, habría que hacer **private** todos los constructores y proporcionar uno o más métodos especiales para crear objetos. De esa forma, estos métodos pueden restringir el número de objetos que se crean y las condiciones en las que se crean. (Un caso particular de esto es el patrón *singleton* que aparece en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*.)

He aquí un ejemplo que muestra las varias formas de implementar el clonado, y luego, cómo ser "deshabilitado" más abajo en la jeraquía:

```
//: apendicea:ComprobarCloneable.java
// Comprobar si se puede clonar una referencia.

// No se puede clonar porque no
// superpone clone():
class Ordinario {}

// Superpone clone, pero no implementa
// Cloneable:
class ClonErroneo extends Ordinario {
  public Object clone()
      throws CloneNotSupportedException {
    return super.clone(); // Lanza excepcion
  }
}

// Hace todo lo necesario para el clonado:
class EsClonable extends Ordinario
    implements Cloneable {
  public Object clone()
      throws CloneNotSupportedException {
    return super.clone();
  }
}

// Desconectar el clonado lanzando la excepción:
class NoMas extends EsClonable {
```

```
  public Object clone()
      throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}

class ProbarMas extends NoMas {
  public Object clone()
      throws CloneNotSupportedException {
    // Llama a NoMas.clone(), lanza excepción:
    return super.clone();
  }
}

class Retornar extends NoMas {
  private Retornar duplicar(Retornar b) {
    // Hacer de alguna forma una copia de b
    // y devolverla. Ésta es una copia estúpida
    // simplemente para que se vea:
    return new Retornar();
  }
  public Object clone() {
    // No llama a NoMas.clone():
    return duplicar(this);
  }
}

// No se puede heredar de éste, así que no
// puede superponer el método clone como en Retornar:
final class RealmenteNoMas extends NoMas {}

public class ComprobarCloneable {
  static Ordinario intentarClonar(Ordinario ord) {
    String id = ord.getClass().getName();
    Ordinario x = null;
    if(ord instanceof Cloneable) {
      try {
        System.out.println("Intentando " + id);
        x = (Ordinario)((esClonable)ord).clone();
        System.out.println("Clonado " + id);
      } catch(CloneNotSupportedException e) {
        System.err.println("No se pudo clonar "+id);
      }
    }
    return x;
```

```
  }
  public static void main(String[] args) {
    // Molde hacia arriba:
    Ordinary[] ord = {
      new EsClonable(),
      new ClonErroneo(),
      new NoMas(),
      new ProbarMas(),
      new Retornar(),
      new RealmenteNoMas(),
    };
    Ordinario x = new Ordinario();
    // Esto no compilara, pues clone() es
    // protected en Object:
    //! x = (Ordinario)x.clone();
    // intentarClonar() Primero comprueba si una
    // clase implementa Cloneable:
    for(int i = 0; i < ord.length; i++)
      intentarClonar(ord[i]);
  }
} ///:~
```

La primera clase, **Ordinario**, representa los tipos de clases que hemos venido viendo a lo largo de todo el libro: sin soporte para el clonado, pero como se ve, tampoco hay ninguna prevención contra el mismo. Si se tiene una referencia a un objeto **Ordinario** que podría haber sufrido una conversión hacia arriba desde una clase aún más derivada, no puede decirse si puede o no ser clonada.

La clase **ClonErroneo** muestra una forma incorrecta de implementar el clonado. Superpone **Object.clone( )** y convierte en **public** ese método, pero no implementa **Cloneable**, por lo que cuando se llama a **super.clone( )** (lo que a efectos es una llamada a **Object.clone( )**), se lanza **CloneNotSupportedException** de forma que el clonado no funcionará.

En **EsClonable** puede verse cómo se llevan a cabo las acciones pertinentes para el clonado: se superpone **clone( )** y se implementa **Cloneable**. Sin embargo, este método **clone( )** y otros muchos que siguen este ejemplo, *no* capturan **CloneNotSupportedException**, sino que en su lugar se lo pasan al llamador, que deberá envolverlo con un bloque *try-catch*. En tus métodos **clone( )**, generalmente tendrás que capturar **CloneNotSupportedException** *dentro* de **clone( )** en vez de pasarlo. Como se verá, en este ejemplo es más informativo pasar las excepciones.

La clase **NoMas** intenta "desactivar" el clonado de la forma que pretendían los diseñadores de Java: en la clase derivada **clone( )** se lanza **CloneNotSupportedException**. El método **clone( )** de la clase **NoMas** llama adecuadamente a **super.clone( )**, y éste resuelve a **NoMas.clone( )**, que lanza una excepción y evita el clonado.

Pero ¿qué ocurre si el programador no sigue la pauta "adecuada" de llamar a **super.clone( )** dentro del método **clone( )** superpuesto? En **Retornar**, puede verse cómo puede ocurrir esto. Esta clase usa un método **duplicar( )** separado para hacer una copia del objeto actual y llama a este método

dentro de **clone( )** *en vez de* invocar a **super.clone( )**. La excepción no se lanza nunca y la clase nueva es clonable. No se puede confiar en que se lance una excepción para evitar hacer clonable una clase. La única solución a prueba de bombas es la mostrada en **RealmenteNoMas**, que es **final**, y de la que por consiguiente no se puede heredar. Esto significa que si **clone( )** lanza una excepción en la clase **final**, no puede ser modificada con herencia y se asegura prevenir el clonado. (No se puede invocar explícitamente a **Object.clone( )** desde una clase que tiene un nivel de herencia arbitrario; uno esta limitado a invocar a **super.clone( )** que tiene acceso sólo a la clase base directa.) Por consiguiente, si se construye cualquier objeto que involucre aspectos de seguridad, se deseará que esas clases sean **final**.

El primer método que se ve en la clase **ComprobarCloneable** es **intentarClonar()**, que toma un objeto **Ordinario** y comprueba si es clonable o no con **instanceof**. Si lo es, convierte el objeto a un **EsClonable**, llama a **clone( )** y convierte el resultado de nuevo a **Ordinario**, capturando cualquier excepción que se lance. Nótese el uso de la identificación de tipos en tiempo de ejecución (ver Capítulo 12) para imprimir el nombre de la clase de forma que pueda verse lo que está ocurriendo.

En **main( )**, se crean distintos tipos de objetos **Ordinario** que son convertidos a **Ordinario** en la definición del array. Las dos primeras líneas de código siguientes crean un objeto **Ordinario** tal cual e intentan clonarlo. Sin embargo, este código no compilará pues **clone( )** es un método **protected** en **Object**. El resto del código recorre el array e intenta clonar cada objeto, informando del éxito o fracaso de cada caso. La salida es:

```
Intentando EsClonable
Clonado EsClonable
Intentando NoMas
No se pudo clonar NoMas
Intentando ProbarMas
No se pudo clonar ProbarMas
Intentando Retornar
Clonado Retornar
Intentando RealmenteNoMas
No se pudo clonar RealmenteNoMas
```

Por tanto, para resumir, si se desea que una clase sea clonable:

1. Implementar la interfaz **Clonable**.
2. Superponer **clone( )**.
3. Invocar a **super.clone()** dentro del **clone( )** propio.
4. Capturar las excepciones dentro del **clone( )** propio.

Esto producirá los efectos más convenientes.

El constructor de copia

El clonado puede parecer un proceso complicado de configurar. Podría parecer que debería haber alguna alternativa. Un enfoque que se le podría ocurrir a alguien (especialmente si se trata de un

programador de C++) es hacer un constructor especial cuyo trabajo sea duplicar un objeto. En C++, a esto se le llama un *constructor de copia*. A primera vista, ésta parece una solución obvia, pero de hecho, no funciona. He aquí un ejemplo:

```
//: apendicea:ConstructorCopia.java
// Un constructor para copiar un objeto del mismo tipo,
// como un intento de crear una copia local.

class CualidadesFruta {
  private int peso;
  private int color;
  private int solidez;
  private int madurez;
  private int olor;
  // etc.
  CualidadesFruta() { // Constructor por defecto
    // Hacer algo que tenga sentido...
  }
  // Otros constructores:
  // ...
  // Constructor de copia:
  CualidadesFruta(CualidadesFruta f) {
    peso = f.peso;
    color = f.color;
    solidez = f.solidez;
    madurez = f.madurez;
    olor = f.olor;
    // etc.
  }
}

class Semilla {
  // Miembros...
  Semilla() { /* Constructor por defecto */ }
  Semilla(Semilla s) { /* Constructor de copia */ }
}

class Fruta {
  private CualidadesFruta cf;
  private int semillas;
  private Semilla[] s;
  Fruta(CualidadesFruta c, int conteoSemilla) {
    cf = c;
    Semillas = conteoSemillas;
    s = new Semilla[semillas];
    for(int i = 0; i < semillas; i++)
```

```
      s[i] = new Semilla();
  }
  // Otros constructores:
  // ...
  // Constructor de copia:
  Fruta(Fruta f) {
    cf = new CualidadesFruta(f.cf);
    semillas = f.semillas;
    // Llamar a todos los constructores de copia de Semilla:
    for(int i = 0; i < semillas; i++)
      s[i] = new Semilla(f.s[i]);
    // Otras actividades de construccion de copias...
  }
  // Para permitir a los constructores derivados (u otros
  // métodos) poner distintas calidades:
  protected void aniadirCualidades(CualidadesFruta c) {
    cf = c;
  }
  protected CualidadesFruta obtenerCualidades() {
    return cf;
  }
}

class Tomate extends Fruta {
  Tomate() {
    super(new CualidadesFruta(), 100);
  }
  Tomate(Tomate t) { // Constructor de copia
    super(t); // Molde hacia arriba para los constructores de copia base
    // Otras actividades de construcción de copias...
  }
}

class CualidadesZebra extends CualidadesFruta {
  private int rayas;
  CualidadesZebra() { // Constructor por defecto
    // hacer algo que tenga sentido...
  }
  CualidadesZebra(CualidadesZebra z) {
    super(c);
    rayas = c.rayas;
  }
}

class ZebraVerde extends Tomate {
```

```
  ZebraVerde() {
    aniadirCualidades(new CualidadesZebra());
  }
  ZebraVerde(ZebraVerde z) {
    super(z); // Invoca Tomate(Tomate)
    // Restaurar las actividades correctas:
    aniadirCualidades(new CualidadesZebra());
  }
  void evaluar() {
    CualidadesZebra cz =
      (CualidadesZebra)obtenerCualidades();
    // Hacer algo con las cualidades
    // ...
  }
}

public class ConstructorCopia {
  public static void madurar(Tomate t) {
    // Usar el constructor de copia:
    t = new Tomate(t);
    System.out.println("En madurar, t es un " +
      t.getClass().getName());
  }
  public static void cortar(Fruta f) {
    f = new Fruta(f); // Ummm... ¿funcionara esto?
    System.out.println("En cortar, f es un " +
      f.getClass().getName());
  }
  public static void main(String[] args) {
    Tomate tomate = new Tomate();
    madurar(tomate); // OK
    cortar(tomate); // OOPS!
    ZebraVerde z = new ZebraVerde();
    madurar(z); // OOPS!
    cortar(z); // OOPS!
    z.evaluar();
  }
} ///:~
```

Esto parece un poco extraño a primera vista. Es verdad que *fruta* tiene *cualidades*, pero ¿por qué no poner datos miembro representando las cualidades directamente en la clase **Fruta**? Hay dos razones potenciales. La primera es que se podría querer insertar o modificar las cualidades de forma sencilla. Nótese que **Fruta** tiene un método **protected aniadirCualidades( )** para permitir que las clases derivadas sí lo hagan. (Se podría pensar que lo lógico es tener un constructor **protected** en **Fruta** que tome un parámetro **CualidadesFruta**, pero los constructores no se heredan por lo que

no estaría disponible en clases heredadas de ésta.) Haciendo que las cualidades de la fruta sean una clase separada, se tiene mayor flexibilidad, incluyendo la potestad para cambiar las cualidades a mitad de camino en la vida de un objeto **Fruta** en particular.

La segunda razón para hacer **CualidadesFruta** un objeto independiente es por si se desea añadir cualidades nuevas o cambiar el comportamiento vía la herencia y el polimorfismo. Nótese que en el caso de **ZebraVerde** (que *verdaderamente* es un tipo de tomate —yo los he cultivado y son geniales), el constructor llama a **aniadirCualidades( )** y le pasa un objeto **CualidadesZebra**, que se deriva de **CualidadesFruta**, de forma que se puede adjuntar a la referencia a **CualidadesFruta** en la clase base. Por supuesto, cuando **ZebraVerde** usa el **CualidadesFruta**, debe hacer una conversión del mismo hacia el tipo correcto (como se vio en **evaluar( )**), pero siempre sabe que ese tipo es **CualidadesZebra**.

También se verá que hay una clase **Semilla**, y que **Fruta** (que por definición lleva sus propias *seeds* o semillas[4]) contiene un array de **Semillas**.

Finalmente, fíjese que cada clase tiene un constructor de copia, y que cada uno se encarga de llamar a los constructores de copia correspondientes a la clase base y los objetos miembros para lograr una copia en profundidad. El constructor de copia se prueba dentro de la clase **ConstructorCopia**. El método **madurar( )** toma un parámetro **Tomate** y hace una construcción de una copia del mismo para duplicar el objeto:

```
t = new Tomate(t);
```

mientras que **cortar( )** toma un objeto **Fruta** más genérico, duplicándolo también:

```
f = new Fruta(f);
```

Éstos se prueban en **main( )** condistintos tipos de **Fruta**. He aquí la salida:

```
En madurar, t es un Tomate
En cortar, f es una Fruta
En madurar, t es un Tomate
En cortar, f es una Fruta
```

Es aquí donde aflora el problema. Tras la construcción de copia que se da al **Tomate** dentro de **cortar( )**, el resultado deja de ser un objeto **Tomate** para ser simplemente una **Fruit**. Ha perdido todas las características que lo hacían ser un Tomate. Y lo que es más, al tomar una **ZebraVerde**, tanto **madurar( )** como **cortar( )** la convierten en un **Tomate** y una **Fruta** respectivamente. Por consiguiente, desgraciadamente, el esquema del constructor de copia no es bueno en Java cuando lo que se pretende es hacer una copia local de un objeto.

¿Por qué funciona en C++ y no en Java?

El constructor de copia es una parte fundamental de C++, puesto que hace automáticamente una copia de un objeto. No obstante, el ejemplo anterior muestra que no funciona en Java. ¿Por qué? En

---

[4] Excepto el pobre aguacate, que ha sido reclasificado a simplemente "grasa".

Java todo lo que manipulamos son referencias, mientras que en C++ se puede tener entidades que parecen referencias y se puede *también* pasar los objetos directamente. Esto es para lo que sirve este constructor: cuando se desea tomar un objeto y pasarlo por valor, duplicando por consiguiente el objeto. Por tanto, funciona bien en C++, pero debería tenerse en cuenta que este esquema falla en Java, por lo que no debe usarse en este lenguaje.

Clases de sólo lectura

Mientras que la copia local producida por **clone( )** da los resultados deseados en los casos apropiados, es un ejemplo de obligar al programador (el autor del método) a responsabilizarse de prevenir los efectos negativos del uso de alias. ¿Qué ocurre si se está construyendo una biblioteca de propósito tan general y uso tan frecuente que no se puede suponer que las operaciones de clonado se harán siempre en los lugares adecuados? O más probablemente, ¿qué ocurre si se *desea* permitir el *uso de alias* para lograr eficiencia —para evitar la duplicación innecesaria de objetos—, pero no se desean los tan negativos efectos laterales del uso de alias?

Una solución es crear *objetos inmutables* que pertenezcan a clases de sólo lectura. Se puede definir una clase de forma que ningún método de la misma cause cambios al estado interno del objeto. En una clase así, el uso de alias no tiene impacto puesto que sólo se puede *leer* el estado interno, de forma que muchos fragmentos de código puedan estar leyendo el mismo objeto sin problemas.

Como ejemplo simple de los objetos inmutables, la biblioteca estándar de Java contiene las clases "envoltorio" para todos los tipos primitivos. Se podría ya haber descubierto que, si se desea almacenar un **int** dentro de un contenedor como una **ArrayList** (que sólo guarda **referencias a Object**), se puede envolver el **int** dentro de la clase **Integer** de la biblioteca estándar:

```
//: apendicea:EnteroInmutable.java
// No se puede alterar la clase Integer.
import java.util.*;

public class EnteroInmutable {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new Integer(i));
    // ¿Pero cómo se cambia el int
    // interno al Integer?
  }
} ///:~
```

La clase **Integer** (además de todas las clases "envoltorio" primitivas) implementa la inmutabilidad de forma simple: no tienen métodos que permitan modificar el objeto.

Si se desea un objeto que guarde un tipo primitivo que se pueda modificar, hay que crearlo a mano. Afortunadamente, esto es trivial:

```
//: apendicea:EnteroMutable.java
// Una clase envoltorio modificable.
import java.util.*;

class ValorInt {
  int n;
  ValorInt(int x) { n = x; }
  public String toString() {
    return Integer.toString(n);
  }
}

public class EnteroMutable {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new ValorInt(i));
    System.out.println(v);
    for(int i = 0; i < v.size(); i++)
      ((ValorInt)v.get(i)).n++;
    System.out.println(v);
  }
} ///:~
```

Nótese que **n** es amigo para simplificar la codificación.

ValorInt puede incluso ser más simple si la inicialización a cero por defecto es la adecuada (momento en el que el constructor deja de ser necesario) y no hay que preocuparse de su impresión (por lo que no se necesita el **toString( )**).

```
class ValorInt ( int n; )
```

Coger el elemento y convertirlo es un poco molesto, pero eso es cosa de **ArrayList**, no de **ValorInt**.

Creando clases de sólo lectura

Es posible crear clases de sólo lectura. He aquí un ejemplo:

```
//: apendicea:Inmutable1.java
// Objetos que no pueden ser modificados
// e inmunes al uso de alias.

public class Inmutable1 {
  private int datos;
  public Inmutable1(int valIni) {
    datos = valIni;
```

```
  }
  public int leer() { return datos; }
  public boolean nocero() { return datos != 0; }
  public Inmutable1 cuadruple() {
    return new Inmutable1(datos * 4);
  }
  static void f(Inmutable1 i1) {
    Inmutable1 cuad = i1.cuadruple();
    System.out.println("i1 = " + i1.leer());
    System.out.println("cuad = " + cuad.leer());
  }
  public static void main(String[] args) {
    Inmutable1 x = new Inmutable1(47);
    System.out.println("x = " + x.leer());
    f(x);
    System.out.println("x = " + x.leer());
  }
} ///:~
```

Todos los datos son **private**, y se verá que ninguno de los métodos **public** modifica esos datos. De hecho, el método que parece modificar un objeto es **cuadruple( )**, pero éste crea un nuevo objeto **Inmutable1** y deja el original intacto.

El método **f( )** toma un objeto **Inmutable1** y lleva a cabo varias operaciones sobre él mismo, y la salida de **main( )** demuestra que no se hace ningún cambio a **x**. Por consiguiente, el objeto de **x** podría recibir tantos alias como se desee sin que esto suponga daño alguno, pues la clase **Inmutable1** está diseñada para garantizar que los objetos no se modifiquen.

Los inconvenientes de la inmutabilidad

Crear una clase inmutable parece en primera instancia proporcionar una solución elegante. Sin embargo, siempre que se necesita un objeto modificado del tipo nuevo, hay que sufrir la sobrecarga debida a la creación del nuevo objeto, además de causar potencialmente más recolecciones de basura. En algunas clases esto no es un problema, pero en otras (como la clase **String**) es prohibitivamente caro.

La solución es crear una clase amiga que *puede* modificarse. Así, al hacer muchas modificaciones, se puede pasar a usar la clase amiga modificable y volver a la clase inmutable cuando se haya acabado.

El ejemplo de arriba puede modificarse así:

```
//: apendicea:Immutable2.java
// Una clase clase amiga para hacer
// cambios a objetos inmutables.
```

```
class Mutable {
  private int datos;
  public Mutable(int valIni) {
    datos = valIni;
  }
  public Mutable sumar(int x) {
    datos += x;
    return this;
  }
  public Mutable multiplicar(int x) {
    datos *= x;
    return this;
  }
  public Inmutable2 hacerInmutable2() {
    return new Inmutable2(datos);
  }
}

public class Inmutable2 {
  private int datos;
  public Inmutable2(int valIni) {
    datos = valIni;
  }
  public int leer() { return datos; }
  public boolean nocero() { return datos != 0; }
  public Inmutable2 sumar(int x) {
    return new Inmutable2(datos + x);
  }
  public Inmutable2 multiplicar(int x) {
    return new Inmutable2(datos * x);
  }
  public Mutable hacerMutable() {
    return new Mutable(datos);
  }
  public static Inmutable2 modificar1(Inmutable2 y){
    Inmutable2 val = y.sumar(12);
    val = val.multiplicar(3);
    val = val.sumar(11);
    val = val.multiplicar(2);
    return val;
  }
  // Esto produce el mismo resultado:
  public static Inmutable2 modificar2(Inmutable2 y){
    Mutable m = y.hacerMutable();
    m.sumar(12).multiplicar(3).sumar(11).multiplicar(2);
```

```
    return m.hacerInmutable2();
  }
  public static void main(String[] args) {
    Inmutable2 i2 = new Inmutable2(47);
    Inmutable2 r1 = modificar1(i2);
    Inmutable2 r2 = modificar2(i2);
    System.out.println("i2 = " + i2.leer());
    System.out.println("r1 = " + r1.leer());
    System.out.println("r2 = " + r2.leer());
  }
} ///:~
```

Inmutable2 contiene métodos que, como antes, preservan la inmutabilidad de los objetos produciendo nuevos objetos siempre que se desee una modificación. Se trata de los métodos **sumar( )** y **multiplicar( )**. La clase amiga se llama **Mutable** y también tiene métodos **sumar( )** y **multiplicar( )**, que modifican el objeto **Mutable** en vez de construir uno nuevo. Además, **Mutable** tiene un método que usa sus datos para producir un objeto **Inmutable2** y viceversa.

Los dos métodos estáticos **modificar1( )** y **modificar2( )** muestran dos enfoques distintos para producir el mismo resultado. En **modificar1( )**, todo se hace dentro de la clase **Inmutable2** y puede verse que en el proceso se crean cuatro objetos **Immutable2** nuevos. (Y cada vez que se reasigna **val**, el objeto anterior se convierte en basura.)

En el método **modificar2( )** puede verse que lo primero que se hace es tomar **Inmutable2 y** y producir un **Mutable** a partir de él mismo. (Esto es simplemente como llamar a **clone( )** como se vio anteriormente, pero esta vez se crea un tipo de objeto distinto.) Después se usa el objeto **Mutable** para llevar a cabo muchas operaciones de cambio *sin* precisar la creación de muchos objetos nuevos. Finalmente, se vuelve a convertir en **Inmutable2**. Aquí, se crean dos objetos nuevos (el **Mutable** y el resultado **Inmutable2**) en vez de cuatro.

Por tanto, este enfoque tiene sentido cuando:

1. Se necesitan objetos inmutables y
2. A menudo son necesarias muchas modificaciones o
3. Es caro crear objetos inmutables nuevos.

Strings inmutables

Considérese el código siguiente:

```
//: apendicea:Encadenador.java

public class Encadenador {
  static String mayusculas(String s) {
    return s.toUpperCase();
```

```
  }
  public static void main(String[] args) {
    String q = new String("hola");
    System.out.println(q); // hola
    String qq = mayusculas(q);
    System.out.println(qq); // HOLA
    System.out.println(q); // hola
  }
} ///:~
```

Cuando se pasa **q** en **mayusculas( )** es de hecho una copia de la referencia a **q**. El objeto al que está conectado esta referencia sigue en una única ubicación física. Se copian las referencias al ser pasadas.

Echando un vistazo a la definición de **mayusculas( )**, se puede ver que la referencia que se pasa es **s**, y que existe sólo mientras se está ejecutando el cuerpo de **mayusculas( )**. Cuando acaba **mayusculas( )**, la referencia local **s** se desvanece. El método **mayusculas( )** devuelve el resultado, que es la cadena de texto original con todos los caracteres en mayúsculas. Por supuesto, de hecho devuelve una referencia al resultado. Pero resulta que esa referencia que devuelve apunta a un nuevo objeto, quedando el **q** original de lado. ¿Cómo ocurre esto?

Constantes implícitas

Si se dice:

```
String s = "asdf";
String x = Encadenador.mayusculas(s);
```

¿se desea verdaderamente que el método **mayusculas( )** *modifique* el parámetro? En general, no, porque un parámetro suele parecer al lector del código como un fragmento de información proporcionado al método, no algo que pueda modificarse. Ésta es una garantía importante, puesto que hace que el código sea más fácil de leer y entender.

En C++, disponer de esta garantía era lo suficientemente importante como para incluir la palabra clave **const**, que permitía al programador asegurar que no se pudiera usar una referencia (en C++, puntero o referencia) para modificar el objeto original. Entonces, se pedía al programador de C++ que fuera diligente y usara **const** en todas partes. Esto puede confundir, además de ser fácil de olvidar.

Sobrecarga de "+" y el **StringBuffer**

Los objetos de la clase **String** están diseñados para ser inmutables, usando la técnica recién mostrada. Si se examina la documentación en línea de la clase **String** (como se resumirá más adelante en este capítulo), se verá que todo método de la clase que sólo aparenta modificar un **String** verdaderamente crea y devuelve un objeto **String** completamente nuevo que contiene la modificación. El **String** original queda intacto. Por consiguiente, no hay una faceta en Java que, como **const** en C++, permita al compilador soportar la inmutabilidad de los objetos. Si se desea lograrla, hay que implementarla a mano, como hace **String**.

Dado que los objetos **String** son inmutables, se pueden establecer tantos alias a un **String** como se desee. Dado que es de sólo lectura, no se puede dar el caso de que una referencia altere algo que afecte a otras. Por tanto, un objeto de sólo lectura soluciona de forma elegante el problema del uso de alias.

También parece posible manejar todas las clases en las que se necesita un objeto modificado, creando una versión completamente nueva del objeto con sus modificaciones, como hace **String**. Sin embargo, para algunas operaciones, esto no es eficiente. En este sentido, destaca el operador "+", sobrecargado para objetos **String**. La sobrecarga implica que se le da un significado extra cuando se usa con cierta clase en concreto. (Los operadores "+" y "+=" para **String** son los únicos sobrecargados en Java, y Java no permite al programador sobrecargar ninguno más)[5].

Cuando se usa con objetos **String**, el "+" permite concatenar **Strings**:

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

Puede imaginarse cómo *debería* funcionar esto: el **String** "abc" podría tener un método **append( )** que creara un nuevo objeto **String** que contuviera "abc" concatenado con los contenidos de **foo**. El nuevo objeto **String** crearía a continuación otro **String** con la adición de "def" y así sucesivamente.

Esto funcionaría, pero requiere de la creación de muchos objetos **String** simplemente para componer este nuevo **String**, y después hay un conjunto de objetos **String** intermedios que deberían ser eliminados por el recolector de basura. Sospecho que los diseñadores de Java intentaron primero este enfoque (que no es sino una lección en diseño de software —no se sabe nada sobre un sistema hasta que se prueba a codificar en él y se tiene algo funcionado). También sospecho que descubrieron que implicaba un rendimiento inaceptable.

La solución es una clase amiga mutable semejante a la ya vista. En el caso de **String**, esta clase compañero se llama **StringBuffer**, y el compilador crea automáticamente un **StringBuffer** para evaluar ciertas expresiones, en particular cuando se usan los operadores sobrecargados + y += con objetos **String**. El ejemplo muestra lo que ocurre:

```
//: apendicea:CadenasInmutables.java
// Demostrando StringBuffer.

public class CadenasInmutables {
  public static void main(String[] args) {
    String foo = "foo";
    String s = "abc" + foo +
```

[5] C++ permite al programador sobrecargar operadores según desee. Debido a que éste puede ser un proceso complicado (ver Capítulo 10 de *Thinking in C++, 2nd edition*, Prentice-Hall, 2000), los diseñadores de Java lo consideraron una faceta "negativa" que no debería incluirse en este lenguaje. No fue algo tan malo pues acabaron haciéndolo ellos mismos, e irónicamente, la sobrecarga sería mucho más fácil de usar en C++ que en Java. Esto se puede ver en Phyton (*http://www.Python.org*), que tiene sobrecargado, por ejemplo, el recolector de basura.

```
      "def" + Integer.toString(47);
    System.out.println(s);
    // El "equivalente" usando StringBuffer:
    StringBuffer sb =
      new StringBuffer("abc"); // ¡Crea un String!
    sb.append(foo);
    sb.append("def"); // ¡Crea un String!
    sb.append(Integer.toString(47));
    System.out.println(sb);
  }
} ///:~
```

En la creación del **String s**, el compilador está haciendo el equivalente al código subsecuente que usa **sb**: se crea un **StringBuffer** y se usa **append( )** para añadir nuevos caracteres de forma directa al mismo (en vez de hacer una copia del mismo cada vez). Mientras que esto es más eficiente, merece la pena recalcar que cada vez que se crea una cadena de caracteres entre comillas, como **"abc"** y **"def"**, el compilador las convierte en objetos **String**. Por tanto, puede que se creen más objetos de los esperados, a pesar de la eficiencia lograda con el uso de **StringBuffer**.

Las clases **String** y **StringBuffer**

He aquí un repaso de los métodos disponibles tanto para **String** como para **StringBuffer** de forma que se pueda tener una idea de cómo interactúan. Estas tablas no contienen todos y cada uno de los métodos, sino aquéllos que son importantes en esta discusión. Los métodos sobrecargados vienen resumidos en una única fila.

En primer lugar, la clase **String**:

Método	**Parámetros, Sobrecarga**	**Uso**
boolean	–	–
Constructor	Sobrecargados: Default, **String**, **StringBuffer**, **char**, arrays, byte **arrays.**	Creación de objetos **String**.
length( )		Número de caracteres del **String**.
charAt( )	**int** Indice	El char en cierta posición del **String**.
getChars( ), getBytes()	El principio y el fin del que copiar, el array en el que copiar, un índice al array destino.	Copiar **chars** o **bytes** a un array externo.

Método	Parámetros, Sobrecarga	Uso
toCharArray( )		Produce un **char[]** que contiene los caracteres del **String**.
equals( ), equalsIgnoreCase( )	Un **String** con el que compararse.	Una comprobación de igualdad sobre los contenidos de dos **Strings**.
compareTo( )	Un **String** con el que compararse.	El resultado es negativo, cero o positivo dependiendo del orden lexicográfico del **String** y del parámetro. ¡Distingue mayúsculas y minúsculas!
regionMatches( )	Desplazamiento en este **String**, el otro **String**, y su desplazamiento y longitud para comparar. La sobrecarga añade "ignorar las mayúsculas / minúsculas".	El resultado **boolean** indica si la región coincide.
startsWith( )	**String** con el que podría empezar. La sobrecarga añade desplazamiento al parámetro.	El resultado **boolean** indica si el **String** comienza con el parámetro.
endsWith( )	**String** que podría ser sufijo de este **String.**	El resultado **boolean** indica si el parámetro es o no un sufijo del **String**.
indexOf( ), lastIndexOf( )	Sobrecargado: **char**, **char** e índice de comienzo, **String**, **String** e índice de comienzo.	Devuelve -1 si no se encuentra el parámetro dentro del **String**, de otra forma devuelve el índice en el que comienza el parámetro. **lastIndexOf( )** busca desde atrás hacia delante.
substring( )	Sobrecargado: índice de comienzo, índices de comienzo y final.	**String** que contiene el conjunto de caracteres especificado.

Método	Parámetros, Sobrecarga	Uso
concat( )	El **String** a concatenar.	Devuelve un nuevo objeto **String** conteniendo los caracteres del **String** original seguidos de los caracteres del parámetro.
replace( )	El carácter que buscar, y el nuevo con el que reemplazarlo.	Devuelve un nuevo objeto **String** en el que se han hecho los reemplazos. Usa el **String** viejo si no se encuentra ninguna ocurrencia.
toLowerCase( ), toUpperCase( )		Devuelve un nuevo objeto **String** con todas las letras cambiadas a mayúsculas o minúsculas, devolviendo el original si no se hacen cambios.
trim( )		Devuelve un nuevo objeto **String** quitándole los espacios en blanco del final, o el mismo si no se hacen cambios.
valueOf( )	Sobrecargado: **Object**, **char[]**, **char[]** y desplazamiento y cuenta, **boolean**, **char**, **int**, **long**, **float**, **double**.	Devuelve un **String** que contiene una representación del parámetro en forma de caracteres.
intern( )		Produce una y sólo una referencia a **String** por cada secuencia de caracteres.

Como puede verse, todo método de **String** devuelve cuidadosamente un nuevo objeto **String** cuando es necesario cambiar su contenido. Nótese también que si los contenidos no varían el método, simplemente devuelve una referencia al **String** original. Esto ahorra espacio de almacenamiento y sobrecarga.

He aquí la clase **StringBuffer**:

Método	Parámetros, Sobrecarga	Uso
Constructor	Sobrecargados: Default, longitud del espacio de almacenamiento intermedio a crear, **String** del que crearlo.	Creación de objetos **StringBuffer**.
toString( )		Crea un **String** a partir del **StringBuffer**.
length( )		Número de caracteres del **StringBuffer**.
capacity( )		Devuelve la cantidad de espacios asignados.
ensure-Capacity( )	Entero que indica la capacidad deseada.	Hace que el **StringBuffer** almacene al menos el número de espacios deseado.
setLength( )	Entero que indica la nueva longitud de la cadena de caracteres del espacio de almacenamiento intermedio.	Trunca o expande la cadena de caracteres anterior. Si se expande, se rellena con valores nulos.
charAt( )	Entero que indica la posición del elemento deseado.	Devuelve el **char** que hay en esa posición del espacio de almacenamiento intermedio.
setCharAt( )	Entero que indica la posición del elemento deseado y el nuevo valor **char** para ese elemento.	Modifica el valor de esa posición.
getChars( )	El principio y el final desde el que copiar, el array en el que copiar, un índice al array de destino.	Copia **chars** a un array externo. No hay **getBytes( )** como en **String**.
append( )	Sobrecargado: **Object**, **String**, **char[]**, **char[]** con desplazamiento y longitud, **boolean**, **char**, **int**, **long**, **float**, **double**.	El parámetro se convierte a String y se añade al final del espacio de almacenamiento intermedio actual, incrementándolo si es necesario.

Método	Parámetros, Sobrecarga	Uso
insert( )	Sobrecargado, cada uno con un primer parámetro del desplazamiento en el que empezar a insertar: **Object, String, char[ ], boolean, char, int, long, float, double**.	El segundo parámetro se convierte a String y se inserta en el espacio de almacenamiento intermedio actual a partir del desplazamiento, incrementando el espacio de almacenamiento intermedio si es necesario.
reverse( )		Se invierte el orden de los caracteres en el espacio de almacenamiento intermedio.

El método más comúnmente usado es **append( )**, usado por el compilador al evaluar expresiones **String** que contienen los operadores "+" y "+=". El método **insert( )** tiene una forma similar y ambos métodos llevan a cabo manipulaciones significativas sobre el espacio de almacenamiento intermedio en vez de crear nuevos objetos.

Los **Strings** son especiales

Hasta el momento se ha visto que la clase **String** no es simplemente otra clase en Java. Hay muchos casos especiales en **String**, siendo todos ellos clases predefinidas y fundamentales para Java. Después está el hecho de que una cadena de caracteres entre comillas se convierte en **String**, por parte del compilador, además de los operadores + y +=. En este apéndice se ha visto el caso especial que quedaba: la inmutabilidad construida tan cuidadosamente usando la clase amiga **StringBuffer** y alguna magia extra por parte del compilador.

Resumen

Dado que en Java todo son referencias, y dado que todo objeto se crea en el montículo y es eliminado por el recolector de basura sólo cuando se deja de usar, el sentido de la manipulación de objetos varía, especialmente al pasar y retornar objetos. Por ejemplo, en C y C++, si se desea inicializar algún espacio de almacenamiento en un método, generalmente se pide que el usuario pase la dirección de ese espacio al método. Si no, habría que preocuparse de quién es responsable de destruir ese espacio cuando se acabe con él. Por consiguiente, la interfaz y el entendimiento de esos métodos es más complicado. Pero en Java no hay que preocuparse nunca por la responsabilidad o por si un objeto seguirá existiendo cuando sea necesario, puesto que el propio lenguaje se encarga de todo ello. Se puede crear un objeto en el momento en que se necesite, y no antes, y nunca preocuparse de la mecánica del paso de responsabilidades relativas a ese objeto: simplemente se pasa la referencia. En ocasiones la simplificación que esto proporciona pasa desapercibida, otras veces es abrumadora.

El inconveniente de toda esta magia subyacente es doble:

1. Siempre se tiene una disminución de eficiencia por la gestión extra de memoria (aunque esta disminución puede no ser muy grande), y siempre hay cierta cantidad de incertidumbre sobre el tiempo que puede llevar ejecutar algo (puesto que puede forzarse a que actúe el recolector de basura siempre que a uno le quede poca memoria). En la mayoría de aplicaciones, los beneficios son superiores a los inconvenientes, y en particular, es posible escribir secciones críticas en el tiempo utilizando métodos **native** (ver Apéndice B).

2. Uso de alias: en ocasiones se puede acabar teniendo de forma accidental dos referencias al mismo objeto, lo cual sólo es un problema si se presupone que ambas apuntan a objetos *diferentes*. Es aquí donde es necesario prestar un poco más de atención y, cuando sea necesario, **clone( )** (clonar) un objeto para evitar que otra referencia se sorprenda por un cambio inesperado. De forma alternativa, se puede soportar el uso de alias persiguiendo eficiencia creando objetos inmutables cuyas operaciones pueden devolver nuevos objetos del mismo o de otro tipo, pero nunca cambiar el objeto original, de forma que nadie que haga referencia al mismo perciba los cambios.

Algunos opinan que en Java la clonación es precisamente un alarde de buen diseño, por lo que, en aras de usarlo, implementan su propia versión del clonado[6] no llamando nunca al método **Object.clone( )**, y eliminando así la necesidad de implementar **Cloneable**, y capturar la **CloneNotSupportedException**. Éste es un enfoque bastante razonable y dado que **clone( )** tiene tan poco soporte en la biblioteca estándar de Java, es también bastante seguro. Pero en la medida en que no se invoque a **Object.clone( )** no hay por qué implementar **Cloneable** o capturar la excepción, por lo que esto también parece algo aceptable.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Demostrar un segundo nivel de uso de alias. Crear un método que tome una referencia a un objeto pero no modifique el objeto de la misma. Sin embargo, el método invoca a un segundo método, pasándole la referencia, y este segundo método sí que modifica el objeto.

2. Crear una clase **miCadena** que contenga un objeto **String** que se inicialice en el constructor, usando el parámetro al mismo. Añadir un método **toString( )** y un método **concatenar( )** que añada un objeto **String** a la cadena de caracteres interna. Implementar **clone( )** en **miCadena**. Crear dos métodos **static**, cada uno de los cuales tome como parámetro una referencia **miCadena x**, e invocar a **x.concatenar("prueba")**, pero en el segundo método, llamando primero a **clone( )**. Probar ambos métodos y mostrar los distintos efectos entre sí.

[6] Doug Lea, que me ayudó a resolver este aspecto, me lo sugirió, diciéndome que simplemente crea una función de nombre **duplicate( )** para cada clase.

3. Crear una clase denominada **Pila** que contenga un **int** que es un número de pila (un identificador único). Hacerlo clonable y darle un método **toString( )**. Crear ahora una clase llamada **Juguete** que contenga un array de **Pila** y un **toString( )** que imprima todas sus pilas. Escribir un método **clone()** para **Juguete** que clone automáticamente todos sus objetos **Pila**. Probarlo clonando **Juguete** e imprimiendo el resultado.

4. Cambiar **ComprobarCloneable.java** de forma que todos los métodos **clone( )** capturen la **CloneNotSupportedException** en vez de pasársela al llamador.

5. Utilizando la técnica de la clase amigo mutable, construir una clase inmutable que contenga un **int**, un **double** y un array de **char**.

6. Modificar **Competir.java** para añadir más objetos miembros a las clases **Cosa2** y **Cosa4** y ver si se puede determinar cómo varían los tiempos con la complejidad —si se trata de una relación lineal o parece algo más complicada.

7. A partir de **Serpiente.java**, crear una versión de copia profunda de la serpiente.

8. Heredar un **ArrayList** y hacer que su método **clone( )** lleve a cabo una copia en profundidad.

B: El Interfaz Nativo Java (JNI[1])

El material de este apéndice se incorporó y usó con el permiso de Andrea Provaglio (www.AndreaProvaglio.com).

El lenguaje Java y su API estándar son lo suficientemente ricos como para escribir aplicaciones completas. Pero en ocasiones hay que llamar a código no-Java; por ejemplo, si se desea acceder a aspectos específicos del sistema operativo, interactuar con dispositivos hardware especiales, reutilizar código base preexistente no-Java, o implementar secciones de código críticas en el tiempo.

Interactuar con código no-Java requiere de soporte dedicado por parte del compilador y de la Máquina Virtual, y de herramientas adicionales para establecer correspondencias entre el código Java y el no-Java. La solución estándar para invocar a código no-Java proporcionada por JavaSoft es el *Interfaz Nativo Java*, que se presentará en este apéndice. No se trata de ofrecer aquí un tratamiento en profundidad, y en ocasiones se asume que uno tiene un conocimiento parcial de los conceptos y técnicas involucrados.

JNI es una interfaz de programación bastante rica que permite la creación de métodos nativos desde una aplicación Java. Se añadió en Java 1.1, manteniendo cierto nivel de compatibilidad con su equivalente en Java 1.0: el interfaz nativo de métodos (NMI). NMI tiene características de diseño que lo hacen inadecuado para ser adoptado en todas las máquinas virtuales. Por ello, las versiones futuras del lenguaje podrían dejar de soportar NMI, y éste no se cubrirá aquí.

Actualmente, JNI está diseñado para interactuar con métodos nativos escritos únicamente en C o C++. Utilizando JNI, los métodos nativos pueden:

- Crear, inspeccionar y actualizar objetos Java (incluyendo arrays y **String**s).
- Invocar a métodos Java
- Capturar y lanzar excepciones
- Cargar clases y obtener información de clases
- Llevar a cabo comprobación de tipos en tiempo de ejecución.

Por tanto, casi todo lo que se puede hacer con las clases y objetos ordinarios de Java también puede lograrse con métodos nativos.

---

[1] N. del traductor: En inglés: *Java Native Interface*.

Invocando a un método nativo

Empezaremos con un ejemplo simple: un programa en Java que invoca a un método nativo, que de hecho llama a la función ejemplo **printf( )** de la biblioteca estándar de C.

El primer paso es escribir el código Java declarando un método nativo y sus argumentos:

```
//: apendiceb:MostrarMensaje.java
public class MostrarMensaje {
  private native void MostrarMensaje(String msj);
  static {
    System.loadLibrary("MsgImpl");
    // Truco Linux, si no puedes acceder a la ruta de tu biblioteca
    // configura tu entorno:
    // System.load(
    //   "/home/bruce/tij2/appendixb/MsgImpl.so");
  }
  public static void main(String[] args) {
    MostrarMensaje app = new MostrarMensaje();
    app.MostrarMensaje("Generado con JNI");
  }
} ///:~
```

A la declaración del método nativo sigue un bloque **static** que invoca a **System.loadLibrary( )** (a la que se podría llamar en cualquier momento, pero este estilo es más adecuado). **System.loadLibrary( )** carga una DLL en memoria enlazándose a ella. La DLL debe estar en la ruta de la biblioteca del sistema. La JVM añade la extensión del nombre del archivo automáticamente en función de la plataforma.

En el código de arriba también se puede ver una llamada al método **System.load( )**, marcada como comentario. La trayectoria especificada en este caso es absoluta en vez de radicar en una variable de entorno. Naturalmente, usar ésta última aporta una solución más fiable y portable, pero si no se puede averiguar su valor se puede comentar **loadLibrary( )** e invocar a esta línea, ajustando la trayectoria al directorio en el que resida el archivo en cada caso.

El generador de cabeceras de archivo: javah

Ahora, compile el archivo fuente Java y ejecute **javah** sobre el archivo **.class** resultante, especificando la opción **-jni** (el *makefile* que acompaña a la distribución de código fuente de este libro se encarga de hacer esto automáticamente):

```
javah -jni MostrarMensaje
```

javah lee el archivo de clase Java y genera un prototipo de función en un archivo de cabecera C o C++ por cada declaración de método nativo. He aquí la salida: el archivo fuente **MostrarMensaje.h** (editado de forma que entre en este libro):

```
/* DO NOT EDIT THIS FILE
   - it is machine generated */
#include <jni.h>
/* Header for class MostrarMensaje*/

#ifndef _Included_MostrarMensaje
#define _Included_MostrarMensaje
#ifdef _cplusplus
extern "C" {
#endif
/*
 * Class:       MostrarMensaje
 * Method:      MostrarMensaje
 * Signature:   (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
Java_MostrarMensaje_MostrarMensaje
  (JNIEnv *, jobject, jstring);

#ifdef _cplusplus
}
#endif
#endif
```

Como puede verse por la directiva del preprocesador **#ifdef__cplusplus**, este archivo puede compilarse con un compilador de C o de C++. La primera directiva **#include** incluye **jni.h**, un archivo de cabecera que, entre otras cosas, define los tipos que pueden verse utilizados en el resto del archivo. **JNIEXPORT** y **JNICALL** son macros que se expanden para hacer coincidir directivas específicas de la plataforma. **JNIEnv**, **jobject** y **jstring** son definiciones de tipos de datos JNI, que se explicarán en breve.

Renombrado de nombres y signaturas de funciones

JNI impone una convención de nombres (denominada *renombrado de nombres*) a los métodos nativos. Esto es importante, pues es parte del mecanismo por el que la máquina virtual enlaza las llamadas a Java con métodos nativos. Básicamente, todos los métodos nativos empiezan con la palabra "Java", seguida del nombre de la clase en la que aparece la declaración nativa Java, seguida del nombre del método Java. El carácter "guión bajo" se usa como separador. Si el método nativo Java está sobrecargado, se añade también la signatura de la función al nombre; puede verse la signatura nativa en los comentarios que preceden al prototipo. Para obtener más información sobre *renombrado de nombres* y signaturas de métodos nativos, por favor acudir a la documentación JNI.

Implementando la DLL

En este momento, todo lo que hay que hacer es escribir archivos de código fuente en C y C++ que incluye el archivo de cabecera generado por **javah**, que implementa el método nativo, después compilarlo y generar una biblioteca de enlace dinámico. Esta parte es dependiente de la plataforma. El código de debajo se compila y enlaza a un archivo que en Windows se llama **MsgImpl.dll** y en Unix/Linux **MsgImpl.so** (el *makefile* empaquetado junto con los listados de código contiene los comandos para hacer esto —está disponible en el CD ROM vinculado a este libro, o como descarga gratuita de *www.BruceEcKel.com*):

```
//: apendiceb:MsgImpl.cpp
//# Probado con VC++ & BC++. Hay que ajustar la ruta de
//# los includes para encontrar las cabeceras JNI. Ver
//# el makefile de este capítulo (en el
//# código fuente descargable) si se desea tener un ejemplo.
#include <jni.h>
#include <stdio.h>
#include "MostrarMensaje.b.h"

extern "C" JNIEXPORT void JNICALL
Java_MostrarMensaje_MostrarMensaje(JNIEnv* env,
jobject, jstring jMsg) {
  const char* msg=env->GetStringUTFChars(jMsg,0);
  printf("Piensa en Java, JNI: %s\n", msg);
  env->ReleaseStringUTFChars(jMsg, msg);
} ///:~
```

Los parámetros que se pasan al método nativo son la pasarela que permite la vuelta a Java. El primero, de tipo **JNIEnv**, contiene todos los anzuelos que te permiten volver a llamar a la JVM. (Echaremos un vistazo a esto en la sección siguiente.) El segundo parámetro tiene significado distinto en función del tipo de método. En el caso de métodos no **static** como el del ejemplo de arriba, el segundo parámetro es equivalente al puntero "this" de C++ y similar a **this** en Java: es una referencia al objeto que invocó al método nativo. En el caso de métodos **static**, es una referencia al objeto **Class** en el que está implementado el método.

Los demás parámetros representan los objetos Java que se pasan a la llamada al método nativo. También se pasan tipos de datos primitivos así, pero vienen por valor.

En las secciones siguientes, explicaremos este código mirando a las formas de acceder y controlar la JVM desde dentro de un método nativo.

Accediendo a funciones JNI: el parámetro **JNIEnv**

Las funciones JNI son aquéllas que usa el programador para interactuar con la JVM desde dentro de un método nativo. Como puede verse en el ejemplo de arriba, todo método JNI nativo recibe un parámetro especial en primer lugar: el parámetro **JNIEnv**, que es un puntero a una estructura de datos especial de JNI de tipo **JNIENv_**. Un elemento de la estructura de datos JNI es un puntero a un array generado por la JVM. Cada elemento de este array es un puntero a una función JNI. Las funciones JNI pueden ser invocadas desde el método nativo desreferenciando estos punteros (es más simple de lo que parece). Toda JVM proporciona su propia implementación de las funciones JNI, pero sus direcciones siempre estarán en desplazamientos predefinidos.

A través del parámetro **JNIENv**, el programador tiene acceso a un gran conjunto de funciones. Estas funciones pueden agruparse en las siguientes categorías:

- Obtener información de la versión
- Llevar a cabo operaciones de clase y objetos
- Acceder a campos de instancia y campos estáticos
- Llamar a métodos de instancia y estáticos
- Llevar a cabo operaciones de Strings y arrays
- Generar y gestionar excepciones Java

El número de funciones JNI es bastante grande y no se cubrirá aquí. Sin embargo, mostraré el razonamiento que hay tras el uso de estas funciones. Para obtener información más detallada, consulte la documentación JNI del compilador.

Si se echa un vistazo al archivo de cabecera **jni.h**, se verá que dentro de la condicional de preprocesador **#ifdef__cplusplus**, se define la estructura **JNIEnv_** como una clase cuando se compile por un compilador de C++. Esta clase contiene varias funciones que te permiten acceder a las funciones JNI con una sintaxis sencilla y familiar. Por ejemplo, la línea de código C++ del ejemplo anterior:

```
env->ReleaseStringUTFChars(jMsg, msg);
```

También podría haber sido invocada desde C así:

```
(*env)->ReleaseStringUTFChars(env, jMsg, msg);
```

Se verá que el estilo de C es (naturalmente) más complicado —se necesita una desreferencia doble del puntero **env**, y se debe pasar también el nombre del puntero como primer parámetro a la llamada a la función JNI. Los ejemplos de este apéndice usan el ejemplo de C++.

Accediendo a Strings Java

Como ejemplo de acceso a una función JNI, considérese el código de **MsgImpl.cpp**. Aquí, se usa el argumento **JNIEnv env** para acceder a un **String** Java. Éstos están en formato Unicode, por lo que si se recibe uno y se desea pasarlo a una función no-Unicode (**printf( )**, por ejemplo) primero hay que convertirlo a caracteres ASCII con la función JNI **GetStringUTFChars( )**. Esta función toma un **String** Java y lo convierte a UTF de 8 caracteres. (Estos 8 bits son suficientes para almacenar valores ASCII, o 16 bits para almacenar Unicode. Si el contenido de la cadena de caracteres original sólo estaba compuesta de caracteres ASCII, la cadena resultante también estará en ASCII.)

GetStringUTFChars( ) es una de las funciones miembro de **JNIEnv**. Para acceder a la función JNI usamos la sintaxis típica de C++ para llamar a funciones miembro mediante un puntero. La forma de arriba se usa para acceder a todas las funciones JNI.

Pasando y usando objetos Java

En el ejemplo anterior, se pasaba un **String** al método nativo. También se pueden pasar objetos Java de tu creación al método nativo. Dentro de éste, se puede acceder a los campos y métodos del objeto que se recibió.

Para pasar objetos, puede usarse la sintaxis general de Java al declarar el método nativo. En el ejemplo de abajo, **MiClaseJava** tiene un campo **public** y un método **public**. La clase **UsarObjetos** declara un método nativo que toma un objeto de clase **MiClaseJava**. Para ver si el método nativo manipula su parámetro, se pone el campo **public** del parámetro, se invoca al método nativo, y después se imprime el valor del campo **public**:

```
//: apendiceb:UsarObjetos.java
class MiClaseJava {
  public int unValor;
  public void dividirPorDos() { unValor /= 2; }
}

public class UsarObjetos {
  private native void
    cambiarObjecto(MiClaseJava obj);
  static {
    System.loadLibrary("UsarObjImpl");
    // Truco de Linux, si no se puede lograr la ruta de
    // la biblioteca, configure su entorno:
    // System.load(
    //"/home/bruce/tij2/appendixb/UsarObjImpl.so");
  }
  public static void main(String[] args) {
    UsarObjetos app = new UsarObjetos();
    MiClaseJava unObj = new MiClaseJava();
```

```
      unObj.unValor = 2;
      app.cambiarObjeto(unObj);
      System.out.println("Java: " + unObj.unValor);
   }
} ///:~
```

Tras compilar el código y ejecutar **javah**, se puede implementar el método nativo. En el ejemplo de debajo, una vez que se obtienen el campo y el ID del método, se acceden a través de funciones JNI:

```
//: apendiceb:UsarObjImpl.cpp
//# Probado con VC++ & BC++. Hay que ajustar la ruta de
//# los includes para encontrar las cabeceras JNI. Ver
//# el makefile de este capítulo (en el
//# código fuente descargable) si se desea tener un ejemplo.
#include <jni.h>
extern "C" JNIEXPORT void JNICALL
Java_UsarObjetos_cambiarObjeto(
JNIEnv* env, jobject, jobject obj) {
   jclass cls = env->GetObjectClass(obj);
   jfieldID fid = env->GetFieldID(
      cls, "unValor", "I");
   jmethodID mid = env->GetMethodID(
      cls, "dividirPorDos", "()V");
   int valor = env->GetIntField(obj, fid);
   printf("Nativo: %d\n", valor);
   env->SetIntField(obj, fid, 6);
   env->CallVoidMethod(obj, mid);
   valor = env->GetIntField(obj, fid);
   printf("Nativo: %d\n", valor);
} ///:~
```

Ignorando el equivalente "this", la función C++ recibe un **jobject**, que es el lado nativo de la referencia al objeto Java que le pasamos desde el código Java. Simplemente leemos **unValor**, la imprimimos, cambiamos el valor, se llama al método **dividirPorDos( )** del objeto, y se imprime de nuevo el valor.

Para acceder a un campo o método Java, primero hay que obtener su identificador usando **GetFieldID( )** en el caso de los campos y **GetMethodID( )** en el caso de los métodos. Estas funciones toman el objeto clase, una cadena de caracteres que contiene el nombre del elemento, y otra cadena que proporciona información de tipos: el tipo de datos del campo, o información de signatura en el caso de un método (pueden encontrarse detalles en la documentación de JNI). Estas funciones devuelven un identificador que se usa para acceder al elemento. Este enfoque podría parecer complicado, pero el método nativo desconoce la disposición interna del objeto Java. En su lugar, debe acceder a los campos y métodos a través de índices devueltos por la JVM. Esto permite que distintas JVMs implementen disposiciones internas de objetos distintas sin que esto afecte a los métodos nativos.

Si se ejecuta el programa Java, se verá que el objeto pasado desde el lado Java es manipulado por el método nativo. Pero ¿qué es exactamente lo que se pasa? ¿Un puntero o una referencia Java? Y ¿qué hace el recolector de basura durante llamadas a métodos nativos?

El recolector de basura sigue operando durante la ejecución de métodos nativos, pero está garantizado que no eliminará ningún objeto durante la llamada a un método nativo. Para asegurar esto, se crean previamente *referencias locales*, que son destruidas inmediatamente después de la llamada al método nativo. Dado que su tiempo de vida envuelve la llamada, se sabe que los objetos serán válidos durante toda la llamada al método nativo.

Dado que estas referencias son creadas y destruidas subsecuentemente cada vez que se llama a la función, no se pueden hacer copias locales en los métodos nativos, en variables **static**. Si se desea una referencia que perdure a través de invocaciones a funciones, se necesita una referencia global. Éstas no las crea la JVM, pero el programador puede hacer una referencia global a partir de una local llamando a funciones de JNI específicas. Cuando se crea una referencia global, uno es responsable de la vida del objeto referenciado. La referencia global (y el objeto al que hace referencia) estarán en memoria hasta que el programador libere la referencia explícitamente con la función JNI apropiada. Es semejante al **malloc( )** y **free( )** de C.

JNI y las excepciones Java

Con JNI, pueden lanzarse, capturarse, imprimirse y relanzarse excepciones Java exactamente igual que si se estuviera dentro de un programa Java. Pero depende del programador el invocar a funciones JNI dedicadas a tratar las excepciones. He aquí las funciones JNI para la gestión de excepciones:

- **Throw( )**

  Lanza un objeto excepción existente. Se usa en los métodos nativos para relanzar una excepción.

- **ThrowNew( )**

  Genera un nuevo objeto excepción y lo lanza.

- **ExceptionOccurred( )**

  Determina si se lanzó una excepción aún sin eliminar.

- **ExceptionDescribe( )**

  Imprime una excepción y la traza de la pila.

- **ExcepcionClear( )**

  Elimina una excepción pendiente.

- **FatalError( )**

  Lanza un error fatal. No llega a devolver nada.

Entre éstas, no se puede ignorar **ExceptionOcurred( )** y **ExceptionClear( )**. La mayoría de funciones JNI pueden generar excepciones y no hay ninguna faceta del lenguaje que pueda usarse en el lugar de un bloque *try* de Java, por lo que hay que llamar a **ExceptionOccurred( )** tras cada llamada a función JNI para ver si se lanzó alguna excepción. Si se detecta una excepción, se puede elegir manejarla (y posiblemente relanzarla). Hay que asegurarse, sin embargo, de que la excepción sea siempre eliminada. Esto puede hacerse dentro de la función **ExceptionClear( )** o en alguna otra función si se relanza la excepción, pero hay que hacerlo.

Hay que asegurar que se elimine la excepción, pues si no, los resultados serían impredecibles si se llama a una función JNI mientras está pendiente una excepción. Hay pocas funciones JNI que pueden ser invocadas de forma segura durante una excepción; entre éstas, por supuesto, se encuentran las funciones de manejo de excepciones.

JNI y los hilos

Dado que Java es un lenguaje multihilo, varios hilos pueden invocar a métodos nativos de forma concurrente. (El método nativo podría suspenderse en el medio de su operación al ser invocado por un segundo hilo.) Depende enteramente del programador el garantizar que la llamada nativa sea inmune a los hilos; por ejemplo, no modifica datos compartidos de forma no controlada. Básicamente, se tienen dos opciones: declarar el método nativo como **synchronized**, o implementar alguna otra estrategia dentro del método nativo para asegurar una manipulación de datos concurrentes correcta.

También se podría no pasar nunca el puntero **JNIEnv** por los hilos, puesto que la estructura interna a la que apunta está ubicada en una base hilo a hilo y contiene información que sólo tiene sentido en cada hilo en particular.

Usando un código base preexistente

La forma más sencilla de implementar métodos JNI nativos es empezar a escribir prototipos de métodos nativos en una clase Java, compilar la clase y ejecutar el archivo **.class** con **javah**. Pero ¿qué ocurre si se tiene un código base grande preexistente al que se desea invocar desde Java? Renombrar todas las funciones de las DLLs para que coincidan con la convención de nombres de JNI no es una solución viable. El mejor enfoque es escribir una DLL envoltorio "fuera" del código base original. El código Java llamaría a funciones de esta nueva DLL, que a su vez invoca a funciones de la DLL original. Esta solución no es sólo un rodeo; en la mayoría de casos hay que hacerlo así siempre porque hay que invocar a funciones JNI con referencias a objetos antes de su uso.

Información adicional

Puede encontrarse más material introductorio, incluyendo un ejemplo en C (en vez de en C++) y una discusión de aspectos Microsoft en el Apéndice A de la primera edición de este libro, que puede encontrarse en el CD ROM que acompaña a este libro, o como descarga gratuita de

www.BruceEckel.com. Hay información más extensa disponible en *java.sun.com.* (en el motor de búsqueda, seleccione las palabras clave "native methods" dentro de "training & tutorials"). El Capítulo 11 de *Core Java 2, Volume II,* por Horstmann & Cornell (Prentice-Hall, 2000) da una cobertura excelente a los métodos nativos.

C: Guías de programación Java

Este apéndice contiene sugerencias para ayudar en el diseño de programas de bajo nivel, y en la escritura de código.

Naturalmente, esto son guías, pero no reglas. La idea es usarlas como inspiración, y recordar que hay situaciones ocasionales en las que es necesario vulnerar o romper una regla.

Diseño

1. **La elegancia siempre es rentable**. A corto plazo, podría parecer que lleva mucho tiempo alcanzar una solución totalmente correcta a un problema, pero cuando funciona a la primera y se adapta sencillamente a situaciones nuevas en vez de requerir horas, días o meses de devanarse los sesos, se ve la recompensa (incluso aunque no pueda medirse). Así no sólo se logra un programa que es fácil de construir y depurar, sino que es además fácil de entender y mantener, y es aquí donde radica el valor económico del esfuerzo. Puede ser necesario tener experiencia para llegar a entender este punto, porque puede parecer, en determinados momentos, que no se está siendo productivo al intentar convertir en elegante un fragmento de código. Hay que resistirse a la tendencia a apresurarse; al final, sólo sirve para ralentizar.

2. **Primero hay que hacer que funcione, y después, que sea rápido**. Esto es cierto incluso cuando se está seguro de que determinado fragmento de código es importante y que puede convertirse en un cuello de botella en el sistema. No lo hagas. Primero, intenta que funcione con el diseño más simple posible. Después, si no es lo suficientemente rápido, optimízalo. Casi siempre se descubre que el problema del cuello de botella no era tal. Hay que ahorrar tiempo para lo que es verdaderamente importante.

3. **Recuerde el principio "divide y vencerás"**. Si el problema al que uno se enfrenta es demasiado complicado, puede intentar pensarse en cuál sería el funcionamiento básico del programa, si se dispusiera de un "fragmento mágico" que se encargara de las partes complicadas. Ese "fragmento" es un objeto —escribe el código que usa el objeto, después mira el objeto y encapsula sus partes *complicadas* en otros objetos, etc.

4. **Separe el creador de la clase del usuario de la misma *(programador cliente)*.** El usuario de la clase es el "cliente" y no necesita saber qué es lo que está ocurriendo tras el comportamiento de la clase. El creador de la clase debe ser experto en diseño de clases y escribir la clase de forma que pueda ser usada por los programadores más novatos posible, eso sí, funcionando de forma robusta dentro de la aplicación. El uso de bibliotecas sólo es sencillo si es transparente.

5. **Al crear una clase, intente hacer que los nombres sean tan sencillos que hagan innecesarios los comentarios.** La meta debería ser hacer la interfaz con el programador cliente conceptualmente simple. Para lograr este fin, puede usarse la sobrecarga de métodos cuando sea apropiado para crear una interfaz intuitiva, fácil de usar.

6. **El análisis y diseño deben producir, al menos, las clases del sistema, sus interfaces públicas y sus relaciones con otras clases, especialmente las clases base.** Si tu metodología de diseño produce más que esto, pregúntate a ti mismo si los fragmentos que produce esa metodología tienen valor a lo largo de la vida del programa. Si no lo tienen, mantenerlos supondrá un coste elevado. Los miembros de los equipos de desarrollo tienden a no mantener nada que no contribuya a su productividad; es algo más que probado que hay muchos métodos de diseño que es mejor no utilizar.

7. **Automatice todo.** Escribe primero código de prueba (antes de escribir la clase) y mantenlo con la clase. Automatiza la ejecución de las pruebas a través de un *makefile* o una herramienta semejante. De esta forma, cualquier cambio podrá verificarse automáticamente ejecutando el código de prueba, y se descubrirán errores inmediatamente. Dado que se sabe que se tiene la red de seguridad del marco de trabajo de prueba, será más fácil hacer cambios inmediatamente tras descubrirlos. Recuérdese que las mayores mejoras en lo que a lenguajes se refiere provienen de pruebas preconstruidas, proporcionadas en la forma de comprobación de tipos, manejo de excepciones, etc., pero que estas facetas no llegan más allá. Hay que ir más allá y crear un sistema robusto creando todas las pruebas que verifican facetas específicas de cada clase o programa.

8. **Escriba el código de prueba (antes de escribir la clase) para verificar que el diseño de la clase sea completo.** Si no se puede escribir código de prueba, entonces uno no está seguro de qué apariencia tiene la clase. Además, el escribir código de prueba a menudo ayuda a que afloren facetas y limitaciones adicionales necesarias para la clase —estas facetas y limitaciones no siempre aparecen durante el análisis y diseño. Las pruebas también proporcionan código de ejemplo que muestra cómo usar esa clase.

9. **Todos los problemas del diseño de software pueden simplificarse introduciendo un nivel adicional de indirección.** Esta regla fundamental de ingeniería del software[1], es la base de la abstracción, la faceta principal de la programación orientada a objetos.

10. **Toda indirección debería tener un significado** (hace referencia a la regla 9). Este significado puede ser tan simple como "poner código comúnmente utilizado en un método simple". Si se añaden niveles de indirección (abstracción, encapsulación, etc.) que no tienen significado, su efecto puede ser tan pernicioso como no tener la indirección adecuada.

11. **Haga las clases tan atómicas como sea posible.** Da a cada clase un propósito simple y claro. Si las clases o el diseño del sistema se vuelven demasiado complicados, divide las clases en otras más simples. El indicador más obvio de este aspecto está relacionado con el ta-

[1] Que me explicó a mí Andrew Koeing.

maño: si una clase es grande, seguro que está haciendo demasiadas cosas, y por tanto, tiene más posibilidades de fallar.

Algunas pistas que sugieren el rediseño de una clase son:

1) Una sentencia switch complicada: considérese el uso del polimorfismo.

2) Un conjunto grande de métodos que cubren distintos tipos de operaciones: la solución pasaría por utilizar varias clases.

3) Un conjunto grande de variables miembro relativos a características bastante heterogéneas: la solución pasaría por utilizar varias clases.

12. **Vigile las listas de parámetros largas.** En ocasiones es difícil escribir, leer y mantener llamadas a métodos. En su lugar, debería intentar moverse el método a una clase en la que sea (más) adecuado, y/o pasar objetos como parámetros.

13. **No se repita a sí mismo.** Si un fragmento de código recurre a muchos métodos de clases derivadas, puede ponerse ese código en un único método de una clase base e invocarlo desde los métodos de las clases derivadas. Así no sólo se ahorra espacio de código, sino que se proporciona una propagación más sencilla de los cambios. En ocasiones, descubrir este código común añade funcionalidad de gran valor a la interfaz.

14. **Vigile las sentencias *switch* y las cláusulas *if-else* encadenadas.** Esto suele ser un indicador de *codificación de comprobación de tipos*, lo que significa que se está eligiendo qué código ejecutar a partir de alguna información de tipos (puede que el tipo exacto no sea obvio a la primera). Generalmente este tipo de código puede reemplazarse con herencia y polimorfismo; una llamada a un método polimórfico puede encargarse de la comprobación de tipos, y permitir una extensibilidad más sencilla y de confianza.

15. **Desde el punto de vista del diseño, busque y separe las cosas que varían de los elementos que permanecen invariantes.** Es decir, busca los elementos del sistema que uno podría desear que cambien sin que esto conllevara un rediseño, y encapsula esos elementos en clases. Puede aprenderse mucho más de este concepto en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*.

16. **No extienda la funcionalidad fundamental mediante subclases.** Si un elemento de una interfaz resulta esencial para una clase, debería estar en la clase base y no añadirse durante la derivación. Si se añaden métodos por herencia, probablemente haya que replantearse el diseño.

17. **Menos es más.** Empieza con una interfaz mínima para una clase, tan pequeña y simple como se necesite para solucionar sencillamente el problema, pero no intentes anticiparte a todas las formas en las que *podría* llegar a usarse la clase. Al usar la clase, se descubrirán formas de expandir la interfaz.

    Sin embargo, una vez que una clase esté en uso, no se podrá manipular su interfaz sin molestar al código cliente. Si hay que añadir más métodos, bien; esto no afectará al código más allá de forzar su recompilación. Pero incluso si los métodos nuevos reemplazan la funcionalidad de

los viejos, hay que dejar la interfaz existente a un lado (puede combinarse, si se desea, con la funcionalidad de la implementación subyacente). Si se desea expandir la interfaz de un método existente añadiendo más parámetros, crea un método sobrecargado con estos nuevos parámetros; de esta forma las llamadas existentes al método existente no se verán afectadas.

18. **Lea sus clases en alto para asegurarse de que tienen sentido.** Hay que hacer referencia a la relación entre una clase base y una clase derivada como "es-un" y con los miembros objeto como "tiene-un".

19. **Al decidir entre herencia y composición, pregúntese si necesita hacer conversiones hacia arriba al tipo base**. Si no, es mejor la composición (objetos miembro) antes que la herencia. Esto puede eliminar la necesidad de múltiples tipos de clase base. Si se hereda, los usuarios pensarán que se supone que tendrán que hacer conversiones hacia arriba.

20. **Use miembros de datos para variaciones en valor y superposición de métodos para variaciones de comportamiento**. Es decir, si se encuentra una clase que usa variables de estado junto con métodos que cambian de estado en función de esas variables, probablemente habría que rediseñarla para que exprese las diferencias de comportamiento entre subclases y métodos superpuestos.

21. **Vigile la sobrecarga**. Un método no debería ejecutar condicionalmente código basado en el valor de un parámetro. En este caso, deberían crearse en su lugar dos o más métodos sobrecargados.

22. **Use jerarquías de excepciones** —preferiblemente derivadas de clases específicas apropiadas en la jerarquía de excepciones Java estándares. La persona que capture las excepciones puede así capturar los tipos específicos de excepción seguidos del tipo base. Si se añaden excepciones derivadas nuevas, el código cliente existente seguirá capturando la excepción a través del tipo base.

23. **En ocasiones la agregación simple se encarga del trabajo**. Un "sistema de confort para viajeros" de una línea aérea consiste en elementos inconexos: asiento, aire acondicionado, vídeo, etc., y sigue siendo necesario crear muchos de éstos en un avión. ¿Se hacen miembros privados y se construye una interfaz nueva entera? No —en este caso, los componentes también son parte de la interfaz pública, por lo que habría que crear objetos miembro públicos. Estos objetos tienen sus propias implementaciones privadas, que siguen siendo seguras. Hay que ser conscientes de que la simple agregación no es una solución a utilizar a menudo, pero en ocasiones, se da.

24. **Considere la perspectiva del programador cliente y de la persona que mantiene el código**. Diseñe su clase para que su uso sea tan obvio como sea posible. Anticípese al tipo de cambios que se harán, y diseñe su clase de forma que estos cambios sean sencillos.

25. **Vigile "el síndrome del objeto gigante"**. Éste suele ser una aflicción de los programadores procedurales que son nuevos en POO y que a menudo acaban escribiendo un programa procedimental y pegándolo en uno o dos objetos gigantes. A excepción de los *marcos de trabajo* de aplicación, los objetos representan conceptos de la aplicación, y no la aplicación en sí.

26. **Si hay que hacer algo feo, al menos, concentre la fealdad dentro de una clase**.

27. **Si hay que hacer algo no portable, haga una abstracción de ese servicio y ubíquela dentro de una clase**. Este nivel extra de indirección evita que se distribuya la falta de portabilidad por todo el programa. (Todo queda dentro del Patrón Bridge.)

28. **Los objetos no deberían simplemente guardar datos**. También deberían tener comportamientos bien definidos. (En ocasiones, los "objetos de datos" son apropiados, pero sólo cuando se usan para empaquetar y transportar un grupo de elementos en casos en los que un contenedor generalizado se vuelve inadecuado.)

29. **Elija primero la composición, cuando se trate de crear nuevas clases a partir de las ya existentes**. Sólo se debería usar la herencia si el diseño lo requiere. Si se usa la herencia donde la composición sería suficiente, los diseños se vuelven innecesariamente complicados.

30. **Use la herencia y la superposición de métodos para expresar diferencias en comportamiento, y campos para expresar variaciones de estado**. Un ejemplo extremo de lo que no se debe hacer es heredar de distintas clases para representar los colores en vez de usar un campo "color".

31. **Vigile la *varianza***. Dos objetos semánticamente diferentes pueden tener acciones o responsabilidades idénticas, y hay una tentación natural que intenta convertir a uno en subclase del otro simplemente para beneficiarse de la herencia. A esto se le llama *varianza*, pero no hay una justificación real para forzar una relación superclase/subclase donde no existe. Una solución mejor es crear una clase base general que produzca una interfaz para ambas como clases derivadas —lo cual requiere algo más de espacio, pero sigue ofreciendo los beneficios de la herencia, y probablemente aportará un descubrimiento importante para el diseño.

32. **Vigile la *limitación* durante la herencia.** Los diseños más simples añaden a los heredados nuevas capacidades. Un diseño sospechoso elimina las viejas capacidades durante la herencia sin añadir nuevas. Pero las reglas se hacen para romperlas, y si se está trabajando a partir de una biblioteca de clases vieja habría que reestructurar la jerarquía de forma que la clase nueva encaje donde debería, sobre la clase vieja.

33. **Use patrones de diseño para eliminar "funcionalidades desnudas".** Es decir, si sólo debería crearse un objeto de una clase, no hay por qué escribir un comentario "Hacer sólo uno de éstos". Envuélvalo en un *singleton*. Si se tiene mucho código entremezclado en el programa principal que crea los objetos, busque a un patrón creativo como un método fábrica en el que pueda encapsularse esa creación. Eliminar "funcionalidad desnuda" no sólo hará que el código sea más fácil de entender y mantener, sino que también lo hará más seguro frente a mantenedores "bien-intencionados" que vengan tras de ti.

34. **Vigile la "parálisis del análisis".** Recuerde que hay que avanzar dentro de un proyecto antes de poder saber todo, y que a menudo la forma mejor y más rápida de aprender alguno de los factores desconocidos pasa por avanzar a la siguiente etapa en vez de tratar de adivinarla mentalmente. No se puede saber la solución hasta que se *tenga*. Java tiene cortafuegos preconstruidos; deje que trabajen para ti. Tus fallos en una clase o en un conjunto de clases no destruirán la integridad de todo el sistema.

35. **Cuando se piensa que se tiene un buen análisis, diseño o implementacion, recórralos.** Traiga a alguien a su grupo —no tiene por qué ser un consultor, sino que puede ser cualquiera de otro grupo de la compañía. Repasar el trabajo con un par de ojos frescos puede revelar problemas en una etapa en la que es aún fácil repararlos, cundiendo finalmente este tiempo y coste invertido que el proceso de recorrido.

Implementación

36. **En general, siga las convenciones de codificación de Sun.** Éstas están disponibles en *http://java.sun.com/docs/codeconv/indesc.html* (el código de este libro ha seguido estas convenciones en la medida en que me ha sido posible). Éstas se usan para lo que constituye probablemente el cuerpo de código más grande al que se expondrán la gran mayoría de programadores Java. Si uno se sale de un estilo de codificación de uso general, el resultado final será código más difícil de leer. Sea cual sea la convención de codificación que se decida usar, hay que asegurarse hacerlo de forma consistente en todo el proyecto. Hay una herramienta que reformatea código Java automáticamente en: *http://home.wtal.de/software-solutions/jindent.*

37. **Sea cual sea el estilo de codificación que se use, verdaderamente hay diferencia si el equipo (y aún mejor, si toda la compañía) lo estandariza.** Esto significa que todo el mundo considere que debe ajustarse al estilo —que puede no ser de uno mismo— aunque no esté conforme. El valor de la estandarización es que lleva menos quebraderos de cabeza ajustar el código, por lo que es más fácil centrarse en el significado del código.

38. **Siga las reglas estándares de mayúsculas y minúsculas.** Ponga en mayúscula la primera letra de un nombre de clase. La primera letra de los campos, métodos y objetos (referencias) debería ser minúscula. Todos los identificadores deberían llevar sus palabras juntas, y poner en mayúscula la primera letra de todas las palabras intermedias. Por ejemplo:

    ```
    EstoEsUnNombredeClase
    estoesUnMetodoOCampo
    ```

    Hay que poner en mayúscula *todas* las palabras de los identificadores primitivos **static final** que tengan inicializadores constantes en sus definiciones. Esto indica que son constantes de tiempo de compilación.

    Los paquetes son un caso especial —sus nombres están formados sólo por letras minúsculas, incluso para palabras intermedias. La extensión de dominio (com, org, net, edu, etc.) también deberían ir en minúsculas. (Éste es uno de los cambios introducidos por Java 2 sobre Java 1.1).

39. **No cree sus propios nombres de miembros de datos private "decorados".** Esto se suele ver en forma de caracteres y guiones bajos encadenados. La notación húngara es el peor ejemplo de esto. En ella, se adjuntan caracteres extra que indican el tipo de datos, uso, localización, etc., como si se estuviera escribiendo en lenguaje ensamblador y el compilador no proporcionara asistencia extra de ningún tipo. Estas notaciones son confusas, difíciles de leer y

desagradables de forzar y mantener. Hay que dejar que las clases y los paquetes se encarguen del ámbito de los nombres.

40. **Siga una "forma canónica"** al crear una clase para uso de propósito general. Incluya definiciones de **equals( )**, **hashCode( )**, **toString( )**, **clone( )** (implementa **Cloneable**), e implemente **Comparable** y **Serializable**.

41. **Use las convenciones de nombres "get", "set" e "is" de los JavaBeans**, para los métodos que lean y cambien campos **private**, incluso si no se piensa que en ese momento se esté construyendo un JavaBean. Esto no sólo facilita el uso de la clase como un Bean, sino que es una forma estándar de nombrar a estos tipos de métodos y de esa forma será más fácil de entender por parte del lector.

42. **Por cada clase que crees, considere incluir un static public test( ) que contenga código para probar esa clase.** No es necesario eliminar el código de prueba para usar la clase en un proyecto, y si se hacen cambios se pueden volver a ejecutar las pruebas de forma sencilla. Este código también proporciona ejemplos de cómo usar la clase.

43. **En ocasiones es necesario heredar para poder acceder a miembros protected de la clase base.** Esto puede llevar a la necesidad de múltiples tipos base. Si no se necesita hacer una conversión hacia arriba, derive primero una clase nueva para que lleve a cabo el acceso protegido. Después, convierta esta clase nueva en un objeto miembro de cualquier clase que necesite usarla, en vez de heredarla.

44. **Evite el uso de métodos final sólo por propósitos de eficiencia.** Use **final** sólo cuando se esté ejecutando el programa, no siendo lo suficientemente rápido, y se verá claro que el cuello de botella radica en un método.

45. **Si dos clases están asociadas mutuamente de alguna forma funcional (como los contenedores y los iteradores), intente convertir a una en clase interna de la otra.** Esto no sólo enfatiza la asociación entre clases, sino que permite que se reutilice el nombre de la clase dentro de un único paquete anidándola dentro de otra clase. La biblioteca de contenedores de Java hace esto definiendo una clase interna **Iterator** dentro de cada clase contenedor, proporcionando así a los contenedores una interfaz común. La otra razón por la que usar una clase interna es parte de la implementacion **private**. Aquí, es beneficioso el ocultamiento de la implementación de la clase interna, frente a la asociación y para evitar la contaminación del espacio de nombres descrita líneas más arriba.

46. **Cada vez que se encuentren clases que parezcan estar fuertemente acopladas entre sí, hay que considerarlas mejoras de codificación y mantenimiento que podrían lograrse usando clases internas.** El uso de clases internas no desacoplará las clases, pero hará el acoplamiento más explícito y conveniente.

47. **No caiga en optimización prematura.** Ésta lleva a la locura. En concreto, no se preocupe por la escritura (o la evitación) de métodos nativos, haciendo que algunos métodos sean **final**, o tratando de lograr que el código sea eficiente desde el mismo momento en que se

construye el sistema por primera vez. La meta principal debe ser probar el diseño, a menos que el diseño exija de por sí cierta eficiencia.

48. **Mantenga los ámbitos de las variables tan pequeños como sea posible de forma que la visibilidad y el tiempo de vida de los objetos sea también lo menor posible.** Esto reduce la opción de usar un objeto en el contexto erróneo y de ocultar un error difícil de encontrar. Por ejemplo, suponga que se tiene un contenedor y un fragmento de código que itera por él. Si se copia ese código para ser utilizado en otro contenedor, se podría acabar de forma accidental usando el contenedor viejo como límite superior del nuevo. Si, sin embargo, el contenedor viejo está fuera de ámbito, el error se detectará en tiempo de compilación.

49. **Use los contenedores de la biblioteca estándar de Java.** Sea sensato y habitúase a su uso, y así incrementará enormemente su productividad. Seleccione **ArrayList** para las secuencias, **HashSet** para los conjuntos, **HashMap** para arrays asociativos, y **LinkedList** para pilas (en vez de **Stack**) y colas.

50. **Para que un programa sea robusto, cada componente debe ser robusto.** Utilice las herramientas que le proporciona Java: control de accesos, excepciones, comprobación de tipos, etc., en cada clase que cree. De esa forma se puede pasar al siguiente nivel de abstracción con seguridad, durante la construcción del sistema.

51. **Prefiera los errores en tiempo de compilación a los errores en tiempo de ejecución.** Intente manejar los errores tan cerca del punto en el que ocurren como sea posible. Prefiera manipular el error en ese momento frente a lanzar una excepción. Capture las excepciones en el gestor más cercano con información suficiente para manipularlo. Haga lo que pueda con la excepción en el nivel actual; si eso no soluciona el problema, vuelva a lanzar la excepción.

52. **Vigile las definiciones de métodos largas.** Los métodos deberían ser unidades breves, funcionales que describen e implementan una parte discreta de una interfaz de clase. Un método que sea largo y complicado es difícil y caro de mantener, y está intentando hacer probablemente demasiado él solo. Si se ve un método así, indica que, al menos, debería dividirse en varios métodos. También puede sugerir la creación de una nueva clase. Los métodos pequeños también pueden potenciar la reutilización dentro de la clase. (En ocasiones, los métodos deben ser grandes, si bien siguen haciendo sólo una cosa.)

53. **Mantenga las cosas "tan *private* como sea posible".** Una vez que se hace público cierto aspecto de una biblioteca (un método, una clase, un campo), no se puede quitar. Si se hace, se puede estropear el código existente de alguna persona, forzándole a rediseñarlo y rescribirlo. Si se hace público sólo lo que se debe, se puede cambiar todo el resto con impunidad, y puesto que los diseños tienden a evolucionar, ésta es una libertad considerable. Así, los cambios en la implementacion tendrán un impacto mínimo en las clases derivadas. La privacidad es especialmente importante cuando se trabaja con varios hilos —sólo los campos **private** pueden protegerse del uso no **syncrhonized**.

54. **Use comentarios de forma liberal, y haga uso de la sintaxis de comentarios-documentación javadoc para producir la documentación de sus programas.** Sin embargo, los comentarios deberían añadir significado genuino al código; los comentarios que sólo rei-

teran lo que el código claramente expresa, son molestos. Nótese que simplemente poner nombres de clases y métodos Java con algo de nivel de detalle puede hacer innecesarios muchos comentarios.

55. **Evite el uso de "números mágicos"** —que son números estrechamente vinculados al código. Éstos constituyen una pesadilla si se necesita variarlos, dado que nunca sabe si "100" es el "tamaño del array" o "cualquier otra cosa". En su lugar, pueden crearse constantes con un nombre descriptivo y usar el identificador de la constante por todo el programa. Esto hace que el programa sea más fácil de entender y mantener.

56. **Al crear constructores, considere las excepciones.** En el mejor caso, el constructor no hará nada que lance una excepción. En el escenario inmediatamente mejor, la clase estará compuesta y se habrá heredado sólo de clases robustas, por lo que no será necesaria ninguna eliminación si se lanza una excepción. Si no, habrá que eliminar las clases compuestas internas a una cláusula **finally**. Si un constructor tiene que fallar, la acción apropiada es que lance una excepción, de forma que el llamador no continúe a ciegas, pensando que el objeto se creó correctamente.

57. **Si su clase necesita cualquier eliminación, cuando el programador cliente haya acabado con el objeto, ubicará el código de eliminación en un método único bien definido** —de nombre parecido a **limpiar( )** que sugiere claramente su propósito. Además, ubique un indicador **boolean** en la clase para indicar si se ha eliminado el objeto de forma que **finalize( )** pueda comprobar "la condición de muerte" (ver Capítulo 4).

58. **La responsabilidad de *finalize()* sólo puede ser verificar "la condición de muerte" de un objeto de cara a la depuración.** (Ver Capítulo 4). En casos especiales, podría ser necesario liberar memoria que de otra forma no sería liberada por el recolector de basura. Puesto que puede que no se invoque a éste para su objeto, no se puede usar **finalize( )** para llevar a cabo la limpieza necesaria. Para ello, hay que crear su propio método de "limpieza". En el método **finalize( )** de una clase, compruebe si el objeto se ha eliminado y lance una clase derivada de **RuntimeException**, si no lo ha hecho, para indicar un error de programación. Antes de confiar en un esquema así, hay que asegurarse de que **finalize( )** funcione en tu sistema. (Podría ser necesario invocar a **System.gc( )** para asegurar este comportamiento.)

59. **Si hay que eliminar un objeto (de otra forma que no sea el recolector de basura) dentro de determinado ámbito, use el enfoque siguiente:** inicialice el objeto y, caso de tener éxito, entre inmediatamente en un bloque **try** con una cláusula **finally** que se encargue de la limpieza.

60. **Al superponer *finalize()* durante la herencia, recuerde llamar a *super.finalize( )*.** (Esto no es necesario si la superclase inmediata es **Object**.) Debería llamarse a **super.finalize( )** como última acción del **finalize( )** superpuesto en vez de lo primero, para asegurar que los componentes de la clase base sigan siendo válidos si son necesarios.)

61. **Cuando se está creando un contenedor de objetos de tamaño fijo, transfiéralos a un array** —especialmente si se está devolviendo este contenedor desde un método. Así, se logra el beneficio de la comprobación de tipos en tiempo de compilación de los arrays, y el reci-

piente del array podría no necesitar convertir los objetos del mismo para poder usarlos. Nótese que la clase base de la biblioteca de contenedores, **java.util.Collection**, tiene dos métodos **toArray( )** para lograr esto.

62. **Eliga *interfaces* frente a clases abstractas.** Si se sabe que algo va a ser una clase base, la primera opción debería ser convertirlo en un **interface**, y sólo si uno se ve forzado a tener definiciones de métodos o variables miembros, modificarlo a clase **abstract**. Un **interface** habla de lo que el cliente desea hacer, mientras que una clase tiende a centrase en (o permitir) detalles de implementación.

63. **Dentro de constructores, haga sólo lo que es necesario para dejar el objeto en el estado adecuado.** Evite de forma activa llamar a otros métodos (excepto para los métodos **final**) puesto que estos métodos pueden ser superpuestos por alguien más para producir resultados inesperados durante la construcción. (Ver Capítulo 7 si se desean más detalles.) Los constructores pequeños y simples tienen menos probabilidades de lanzar excepciones o causar problemas.

64. **Para evitar una experiencia muy frustrante, asegúrese de que sólo hay una clase no empaquetada de cada nombre en cualquier parte del classpath.** Si no, el compilador puede encontrar primero la otra clase de nombre idéntico, y pasar mensajes de error que no tienen sentido. Si sospecha tener un problema de *classpath*, intente buscar archivos **.class** con los mismos nombres en cada punto de comienzo del *classpath*. De forma ideal, ponga todas tus clases en paquetes.

65. **Vigile la sobrecarga accidental.** Si se intenta superponer un método de clase base y no se deletrea bien, se puede acabar teniendo un método nuevo en vez de superponer el ya existente. Sin embargo, esto es perfectamente legal, por lo que no se obtendrá ningún mensaje de error por parte del compilador o del sistema de tiempo de ejecución —simplemente, el código no funcionará correctamente.

66. **Vigile la optimización prematura.** Primero haga que funcione, y después, que lo haga rápido —pero sólo si se debe, y sólo si se prueba que hay un cuello de botella de rendimiento en determinada sección del código. A menos que se haya usado un comprobador de eficiencia para descubrir un cuello de botella, probablemente se esté malgastando el tiempo. El coste oculto de problemas de rendimiento es que el código se vuelve menos entendible y mantenible.

67. **Recuerde que el código se lee más de lo que se escribe.** Los diseños limpios facilitan los programas fáciles de entender, pero los comentarios, explicaciones detalladas y ejemplos, tienen un valor incalculable. Todos ellos le ayudan tanto a usted como a cualquiera que venga por detrás. Si no, puede experimentarse a modo de ejemplo la frustración que le depara el intentar entender la documentación en línea de Java.

D: Recursos Software

El JDK de *java.sun.com*. Incluso si se elige un entorno de desarrollo de un tercero, siempre es buena idea tener el JDK a mano por si se presenta algo que pudiera ser un error del compilador. El JDK es la referencia y si hay un fallo en él, seguro que éste es bien conocido.

La documentación HTML de Java de *java.sun.com*. Nunca he encontrado un libro de referencias sobre bibliotecas estándar de Java que no estuviera anticuado o al que le faltara algo. Aunque la documentación HTML de Sun está salpicada de pequeños fallos y en ocasiones es tan tersa que no se puede usar, todas las clases y métodos, al menos *están* ahí. La gente suele mostrarse poco cómoda al principio al usar un recurso en línea en vez de un libro impreso, pero al menos se puede disponer de información a grandes rasgos. Si no es suficiente, entonces deberá acudirse a libros impresos.

Libros

Thinking in Java, 1st Edition. Disponible como HTML completamente indexado, ,y con la sintaxis reemplazada en el CD ROM en este libro, o como descarga gratuita de *www.BruceEckel.com*. Incluye material viejo y material que no se tuvo lo suficientemente en consideración como para incluirlo en la segunda edición.

Core Java 2, de Horstmann & Cornell, Volume I —Fundamentals (Prentice-Hall, 1999). Volume II —Advanced Features, 2000. Libro muy comprensivo y el primer recurso al que acudo yo cuando necesito respuestas. Es el libro que yo recomiendo cuando se ha acabado *Piensa en Java*, y se precisa pasar a un nivel superior.

Java in a Nutshell: A Desktop Quick Reference, 2nd Edition, de David Flanagan *(O'Reilly, 1997)*. Un resumen compacto de la documentación en línea de Java. Prefiero navegar por los documentos de *java.sun.com*, especialmente porque varían a menudo. Sin embargo, tengo muchos colegas que prefieren documentación impresa, y ésta satisface los requisitos; también proporciona mayores discusiones que la documentación en línea.

The Java Class Libraries: An Annotated Reference, de Patrick Chan y Rosanna Lee (Addison-Wesley, 1997). Es lo que *debería* haber sido la documentación en línea: bastante descripción con el propósito de hacer algo usable. Uno de los revisores técnicos de *Piensa en Java* dijo, "si sólo tuviera un libro técnico de Java, sería éste (bien, además del tuyo, claro)". Yo no estoy tan enamorado del libro como él. Es grande, caro y la calidad de sus ejemplos no me satisface. *Pero* es un lugar al que acudir cuando no se queda bloqueado y parece tener más profundidad (y tamaño) que *Java in a Nutshell*.

Java Network Programming, de Elliotte Rusty Harold (O'Reilly, 1997). No empecé a entender la red en Java hasta que encontré este libro. También encuentro su sitio web, Café au Lait, estimulante, digna de mención y una perspectiva actualizada de los desarrollos de Java, y a la que todos los

vendedores veneran. Sus actualizaciones regulares permiten mantenerse al día de las noticias relativas a Java tan rápidamente cambiante. Ver *http://metalab.unc.edu/javafaq/*.

JDBC Database Access with Java, de Hamilton, Catell & Fisher (Addison-Wesley-1997). Si no se sabe nada de SQL y bases de datos, es una introducción buena y gentil. También contiene algunos detalles además de una "referencia anotada" a la API (de nuevo, lo que debería haber sido la documentación en línea). Su pega, como con todos los libros de *The Java Series* ("Los ÚNICOS libros autorizados por JavaSoft") es que ha sido censurado de forma que sólo dice cosas maravillosas sobre Java —nadie puede encontrar pegas en ningún libro de esta serie.

Java Programming with CORBA, de Andreas Vogel & Keith Duddy (John Wiley & Sons, 1997). Un tratamiento serio del tema con ejemplos de código para tres ORBs Java (Visibroker, Orbix, Joe).

Design Patterns, de Gamma, Helm, Jonson & Vlissides (Addison-Wesley, 1995). El primer libro que gestó el movimiento de los patrones en el mundo de la programación.

Practical Algorithms for Programmers, de Binstock & Rex (Addison-Wesley, 1995). Los algoritmos están en C, por lo que son bastante fáciles de traducir a Java. Cada algoritmo incluye una explicación muy detallada.

Análisis y Diseño

Extreme Programming Explained, de Kent Beck (Addison-Wesley, 2000) (próxima publicación en castellano, 2002). *Me encanta* este libro. Sí, tiendo a tomar un enfoque radical de las cosas, pero siempre he pensado que podría haber un proceso de desarrollo de programas diferente y mucho mejor, y creo que XP está bastante próximo. El único libro que ha tenido un impacto comparable en mí fue *PeopleWare* (descrito más abajo), que habla principalmente del entorno y de cómo tratar la cultura corporativa. *Extreme Programming Explained* habla sobre la programación, e incluye todo, incluso los "hallazgos" más recientes. Incluso llegan tan lejos que dicen que los dibujos estén bien siempre y cuando no se gaste demasiado tiempo en ellos, teniendo en mente que acabarán en la basura. (Se verá que este libro *no* tiene el "sello UML de aprobación" en su cubierta). Llegaría a elegir si trabajar en una compañía o no sólo en función de si usa XP. Es un libro pequeño, con capítulos pequeños, cuya lectura no supone un gran esfuerzo, y que presenta temas excitantes. Uno empieza a pensar que trabaja en una atmósfera así e incluso acaba teniendo visiones sobre todo el mundo.

UML Distilled, 2nd Edition, de Margin Fowler (Addison-Wesley, 2000). Cuando se encuentra UML por primera vez, asusta ver tantos diagramas y detalles. De acuerdo con Fowler, la mayoría de esto es innecesario y él corta por lo sano dejando sólo la esencia. Para la mayoría de proyectos, sólo hay que conocer unas pocas herramientas de generación de diagramas, y la meta de Fowler es llegar a un buen diseño en vez de preocuparse por todos los artefactos para llegar a él. Un libro delgado, bueno y legible; el primero que debería consultarse si hubiera que entender UML.

UML Toolkit, de Hans-Erik Eriksson & Magnus Penker (John Wiley & Sons, 1997). Explica UML y cómo usarlo,y tiene un caso de estudio en Java. El CD ROM que adjunta contiene el código Java y una versión limitada de Rational Rose. Una introducción excelente a UML y a cómo usarlo para construir un sistema real.

The Unified Software Development Process, de Ivar Jacobson, Grady Booch y James Rumbaugh (Addison-Wesley, 1999) (disponible en castellano). Mi predisposición hacia este libro era que no me gustara. Parecía tener todos los elementos de un libro de texto aburrido. Me llevé una grata sorpresa —sólo algunas pequeñas partes del libro contienen explicaciones que parece como si los conceptos ni siquiera estuvieran claros para los autores. La gran mayoría del libro no sólo es clara, sino digna de disfrutarla. Y lo mejor de todo, el proceso tiene mucho sentido. No es Programación Extrema (y no tiene su claridad a la hora de las pruebas) pero también es parte de UML —incluso si no se pudiera adoptar XP, la mayoría de gente está de acuerdo en que "UML es bueno" (independientemente del nivel de experiencia que tengan con el mismo) por lo que se podría adoptar. Creo que este libro debería ser el abanderado del UML, y el que debería leerse tras *UML Distilled* de Fowler si se desea más nivel de detalle.

Antes de elegir un método, ayuda a tener perspectivas de aquéllos que no intentan vender ninguno. Es fácil adoptar un método sin entender verdaderamente qué se desea de él o para qué sirve exactamente. Otros lo usan, y eso parece una razón de peso. Sin embargo, las personas tienen un comportamiento psicológico algo extraño en ocasiones: cuando desean creer que algo soluciona sus problemas, lo prueban. (Esto es experimentar, que es bueno). Pero si no soluciona sus problemas, pueden redoblar sus esfuerzos y anunciar a voz en grito lo maravilloso que es lo que han descubierto. Se presupone, en este caso, que si se logra que más gente monte en el mismo barco, uno ya no está sólo, incluso aunque el barco no lleve a ninguna parte (o se esté hundiendo).

No quiero decir que ninguna metodología conduzca a ninguna parte, sino que uno debería pensar en la técnica experimental ("No funciona; vamos a buscar alguna otra cosa") en vez del modo negación ("No, eso no es verdaderamente un problema. Todo va bien, no tenemos por qué cambiar"). Creo que los libros siguientes, leídos *antes* de elegir un método, le proporcionarán bastante ayuda.

Software Creativity, de Robert Glass (Prentice-Hall, 1995). Éste es el mayor libro que he visto que discuta la *perspectiva* de todos los aspectos metodológicos. Es una colección de pequeños ensayos y artículos escritos por Glass, o bien recopilados por él (P. J. Plauger es uno de sus colaboradores), reflejando muchos años de pensamiento y estudio en la materia. Son entretenidos sólo lo suficientemente largos como para decir aquello que es necesario; no se extiende ni aburre. No es simplemente humo, sino que incluye cientos de referencias a otros artículos y estudios. Todo programador y gestor debería leer este libro antes de introducirse en el mar de la metodología.

Software Runaways: Monumental Software Disasters, de Robert Glass (Prentice-Hall, 1997). Lo mejor de este libro es que presenta aquello de lo que no se suele hablar: proyectos no sólo que fallaron, sino que lo hicieron estrepitosamente. Creo que la mayoría de nosotros pensamos "Eso no me puede pasar a mí" (o "Eso no me puede *volver a* ocurrir"), y creo que esto te coloca en clara desventaja. Tener en mente que todo puede fallar te permite estar en una posición mucho mejor como para hacer todo bien.

Peopleware, 2nd Edition, de Tom Demarco y Timothy Lister (Dorset House, 1999). Aunque tienen experiencia en el desarrollo de software, este libro es sobre proyectos y equipos en general. Se centran en la *gente* y sus necesidades, en vez de en la tecnología y sus necesidades. Hablan de la creación de un entorno en el que todo el mundo sea feliz y productivo, en vez de decidir qué reglas deberían seguir estas personas para ser los componentes adecuados de una máquina. Esta última

actitud, creo, es la mayor contribución a la sonrisa y disfrute de los programadores cuando se adopta el método XYZ y simplemente se hace lo que se ha hecho siempre.

Complexity, de M. Mitchell Waldrop (Simon & Schuster, 1992). Se trata de una crónica de cómo varios científicos de varias disciplinas se juntan en Santa Fe, Nuevo Méjico, para discutir problemas reales que no podrían solucionar sus disciplinas individuales (la bolsa en económicas, la formación inicial de la vida en biología, por qué la gente hace lo que hace en sociología, etc.) Atravesando la física, economía, química, matemáticas, computación, sociología, etc. se logra un enfoque multidisciplinar. Pero lo que es más importante, emerge una forma distinta de *pensar* en estos problemas ultra-complejos: lejos del determinismo matemático y de la ilusión con la que se puede escribir una ecuación que prediga todo comportamiento, y más dirigida a *observar* y buscar un posible patrón e intentar emular ese patrón por cualquier medio. (El libro relata, por ejemplo, la emergencia de algoritmos genéticos). Este tipo de pensamiento, creo, es útil puesto que observamos formas de gestionar proyectos software más y más complejos.

Python

Learning Python, de Mark Lutz y David Ascher (O'Reilly, 1999). Una introducción genial para programadores a lo que se está convirtiendo rápidamente en mi lenguaje favorito, un compañero excelente para Java. El libro incluye una introducción a JPython, lo que permite combinar Java y Python en un mismo programa (el intérprete JPython se compila a *bytecodes* Java, así que no hay que añadir nada especial para que todo esto sea posible). Esta unción de lenguajes promete unas posibilidades enormes.

Mi propia lista de libros

Listados en orden de publicación, no todos ellos siguen estando disponibles.

Computer Interfacing with Pascal & C, (Autopublicado por la imprenta Eisys, 1988. Disponible únicamente vía *http://www.BruceEckel.com*). Una introducción a la electrónica desde cuando CP/M seguía siendo el rey y DOS un principiante. Usé lenguajes de programación de alto nivel y muy a menudo el puerto paralelo del ordenador para manejar varios proyectos electrónicos. Adaptación de mis columnas en la primera y mejor revista para la que he escrito, *Micro Cornucopia* (Para parafrasear a Larry O'Brien, editor durante mucho tiempo de *Software Development Magazine*: la mejor revista de informática jamás publicada —incluso tenían ¡planes para fabricar un robot en un florero!.).Micro C se perdió mucho antes de que apareciera Internet. Crear este libro fue una experiencia de publicación muy gratificante.

Using C++, (Osborne/McGraw-Hill, 1993). Uno de los primeros libros de C++. Está fuera de edición y reemplazada por su segunda edición, de nombre *C++ Inside & Out.*

C+ Inside & Out, (Osborne / McGraw-Hill, 1993). Como se ha dicho, es de hecho la 2ª edición de **Using C++**. El C++ de este libro es razonablemente exacto, pero data de 1992, así que *Thinking in*

C++ trata de reemplazarlo. Puede encontrarse más información sobre este libro y descargar el código fuente en *http://www.BruceEckel.com*.

Thinking in C++, 1st edition, (Prentice-Hall, 1995).

Thinking in C++, 2nd Edition, Volume I, (Prentice-Hall, 2000). Descargable de *http://www.BruceEckel.com*.

Black Belt C++, the Master's Collection, Bruce Eckel, editor (M&T Books, 1994). Fuera de tirada. Una colección de capítulos de varias reflexiones sobre C++ basadas en sus presentaciones en la serie de C++ de la *Software Development Conference*, que presidí. La cubierta de este libro me estimuló a intentar controlar futuros diseños de las portadas.

Thinking in Java, 1st Edition, (Prentice-Hall, 1998). La primera edición de este libro ganó el premio a la productividad de la *Software Development Magazine*, el premio *Java Developer's Editor's Choice*, y el premio al mejor libro de *JavaWorld Reader's Choice*. Descargable de *http://www.BruceEckel.com*.

E: Correspondencias español-inglés de clases, bases de datos, tablas y campos del CD ROM que acompaña al libro[1]

Español	Inglés
_TiempoExactoImplBase	_ExactTimeImplBase
_TiempoExactoStub	_ExactTimeStub
A1	A1
AAM	MNA
AbajoL	DownL
AbajoMaxL	DownMaxL
AbrirL	OpenL
AccesoAnidamientoMultiple	MultiNestingAccess
ActorFeliz	HappyActor
ActorTriste	SadActor
Afirmacion	Assert
Ajedrez	Chess
Albaricoque	Apricot
Alias1	Alias1
Alias2	Alias2
Almuerzo	Lunch
AlmuerzoPortable	PortableLunch
AnalizarSentencia	AnalyzeSentence
Anfibio	Amphibian
AniadirClonado	AddingClone
Animacion	Cartoon
Animal	Animal
AnimalDomestico	Pet
AnimalesDomesticos	Pets
ApagarAgua	WaterOff
ApagarLuz	LightOff
Apariencia	LookAndFeel

[1] Apéndice realizado por los profesores Javier Parra, Ricardo Lozano y Luis Joyanes

Español	Inglés
AparienciaModificacionConstante	FinalOverridingIllusion
Applet1	Applet1
Applet1b	Applet1b
Applet1c	Applet1c
Applet1d	Applet1d
Arbol	Tree
Arboles	Trees
Arbusto	Bush
ArchivoEntrada	InputFile
AreaTexto	TextArea
ArrayMultidimensional	MultiDimArray
ArribaL	UpL
ArribaMaxL	UpMaxL
Asignacion	Assignment
Aspersor	SprinklerSystem
Aventura	Adventure
B1	B1
B1L	B1L
B2	B2
B2L	B2L
B3L	B3L
B4L	B4L
Banio	Bath
BarL	BarL
Base	Base
Basura	Garbage
BazL	BazL
BeanExplosion	BangBean
BeanExplosion2	BangBean2
BeanTiempoPerfecto	PerfectTimeBean
BL	BL
Bloqueable	Blockable
Bloqueado	Blocked
Bloqueo	Blocking
Bocadillo	Sandwich
BorderLayout1	BorderLayout1
Bordes	Borders
Boton1	Button1
Boton2	Button2
Boton2b	Button2b
Botones	Buttons
BotonesOpcion	RadioButtons
BotonHTML	HTMLButton
BotonToe	ToeButton

Español	Inglés
Box1	Box1
Box2	Box2
Box3	Box3
Box4	Box4
BoxLayout1	BoxLayout1
BreakYContinue	BreakAndContinue
Buscador	Fetcher
Buscar	Lookup
BuscarAlfabeticamente	AlphabeticSearch
BuscarEnArray	ArraySearching
BuscarL	FetchL
BuscarV	VLookup
CadenasInmutables	ImmutablesStrings
CajaC	CBox
CajaC2	CBox2
CajasColores	ColorBoxes
CajasColores2	ColorBoxes2
CajasMensajes	MessageBoxes
Calc1L	Calc1L
Calc2L	Calc2L
CambiarSystemOut	ChangeSystemOut
Campana	Bell
CamposTexto	TextFields
CapitalesPaises	CountryCapitals
Caracteristica	Characteristic
CaracteristicasExtra	ExtraFeatures
Caramelo	Candy
Caras	Faces
CargarBD	LoadDB
Carrera	Inning
CarreraTormentosa	StormyInning
Casa	House
CasillasVerificacion	CheckBoxes
Cena	Dinner
CerebroPequenio	SmallBrain
Chicle	Gum
CIDSQL	CIDSQL
Cientifico	Scientist
CientificoLoco	MadScientist
Circulo	Circle
CIUDAD	CITY
Clave	Key
ClienteMultiParlante	MultiJabberClient
ClienteParlante	JabberClient

Español	**Inglés**
ClienteTiempoExacto	ExactTimeClient
ClienteTiempoPerfecto	PerfectTimeClient
ClienteTiempoRemoto	RemoteTimeClient
Clonar	Cloning
ClonErroneo	WrongClone
CMIL	CMIL
Coche	Car
Cola	Queue
Coleccion1	Collection1
Colecciones2	Collections2
ColeccionSencilla	SimpleCollection
ColisionInterfaces	InterfaceCollision
Comida	Meal
ComparadorAlfabetico	AlphabeticComparator
ComparadorAlfabetico	AlphabeticComparator
ComparadorTipoComp	CompTypeComparator
CompararArrays	ComparingArrays
Compartiendo1	Sharing1
Compartiendo2	Sharing2
Competir	Compete
Componente	Widget
ComprimirGZIP	GZIPCompress
ComprimirZip	ZipCompress
ComprobarCloneable	CheckCloneable
ConConstantes	WithFinals
CondicionMuerte	DeathCondition
ConectarCID	CIDConnect
ConFinally	WithFinally
Confiteria	SweetShop
CongelarExtraterrestre	FreezeAlien
ConInterna	WithInner
Conjunto1	Conjunto1
Conjunto2	Conjunto2
ConjuntoEventos	EventSet
ConmutadorL	ToggleL
Console	Console
ConstanteBlanca	BlankFinal
ConstructorCopia	CopyConstructor
ConstructoresCompletos	FullConstructors
ConstructoresMultiples	PolyConstructors
ConstructorPorDefecto	DefaultConstructor
ConstructorSimple	SimpleConstructor
ConstructorSimple2	SimpleConstructor2
CONTACTO	CONTACT

Español	Inglés
Contador	Counter
Contador1	Counter1
Contador2	Counter2
Contador3	Counter3
Contador4	Counter4
Contador5	Counter5
Contenidos	Contents
ContextoPagina.jsp	PageContext.jsp
Controlador	Controller
ControlesInvernadero	GreenhouseControls
ConvertirNumeros	CastingNumbers
Cookies.jsp	Cookies.jsp
CopiaLocal	LocalCopy
CopiaProfundidad	DeepCopy
CopiarArrays	CopyingArrays
CopiarL	CopyL
CORREO	EMAIL
CortarL	CutL
CortarYPegar	CutAndPaste
CortoCircuito	ShortCircuit
Cosa1	Thing1
Cosa2	Thing2
Cosa3	Thing3
Cosa4	Thing4
Costumbre	Custom
CP	ZIP
CrearDirectorios	MakeDirectories
CrearTablasCID	CIDCreateTables
CriaturaViviente	LivingCreature
CtlSerial	SerialCtl
Cuadrado	Square
CuadrosCombinados	ComboBoxes
CualidadesFruta	FruitQualities
CualidadesZebra	ZebraQualities
Cuchara	Spoon
Cuchillo	Knife
CuentaString	CountedString
Cuerda	Stringed
Datos	Data
DatosConstante	FinalData
Degradacion	Demotion
DemoExcepcionSencilla	SimpleExceptionDemo
DemoFlujoES	IOStreamDemo
Demonio	Daemon

Español	Inglés
Demonios	Daemons
Derivada	Derived
Desbordamiento	Overflow
DescongelarExtraterrestre	ThawAlien
DespDchaSinSigno	URShift
Destino	Destination
DetectorPrimavera	SpringDetector
DetectorPrimavera2	SpringDetector2
Detergente	Detergent
DIAHORA	DATETIME
Dialogos	Dialogs
DialogoToe	ToeDialog
DibujarOnda	SineWave
DibujarSeno	SineDraw
Dinosaurio	Dinosaur
DIRECCION	ADDRESS
DIRECCIONES	DIRECTIONS
DocumentoMayusculas	UpperCaseDocument
DocumentoMayusculas	UpperCaseDocument
DosContadores	TwoCounter
Dragon	DragonZilla
Durmiente1	Sleeper1
Durmiente2	Sleeper2
Eco	Echo
EcoFormulario	EchoForm
EjecucionFinally	FinallyWorks
Elector	Peeker
Eliminacion	PopFaul
EliminarCalificadores	StripQualifiers
Emergente	Pop-up
Emisor	Sender
EmpezarL	StartL
Encadenador	Stringer
EncenderAgua	WaterOn
EncenderApagarInterruptor	OnOffSwitch
EngendrarDemonio	DaemonSpawn
EnteroInmutable	ImmutableInteger
EnteroMutable	MutableInteger
Enumeraciones	Enumerations
Envoltorio	Wrapping
Equivalencia	Equivalence
ErrorViento	WindError
Escarabajo	Beetle
Escenario	Stage

Español	Inglés
EsClonable	IsCloneable
EscribirDatos	PrintData
Escritura	Printer
EspectaculoDeMiedo	HorrorShow
EsperarNotificar1	WaitNotify1
EsperarNotificar2	WaitNotify2
Estadisticas	Statistics
ESTADO	STATE
EstadoCAD	CADState
Estornudo	Sneeze
Evento	Event
EVENTOS	EVENTS
EventosDinamicos	DynamicEvents
Evolucionado	Further
EVT_ID	EVT_ID
EVTMIEMS	EVTMEMS
ExcepcionBeisbol	BaseballException
ExcepcionCuatro	FourException
ExcepcionEncender1	OnOffException1
ExcepcionEncender2	OnOffException2
ExcepcionMuyImportante	VeryImportantException
ExcepcionSencilla	SimpleException
ExcepcionTormenta	StormException
ExcepcionTres	ThreeException
ExcepcionTrivial	HoHumException
ExploradorClases	ClassScanner
Exterior	Foreign
Extraterrestre	Alien
FalloRapido	FailFast
Falta	Foul
FamiliaVsTipoExacto	FamilyVsExactType
FanEstatico	StaticFun
Figura	Shape
Figuras	Shapes
FiltroDirectorio	DirFilter
FiltroJava	JavaFilter
FL	FL
Flor	Flower
FlowLayout1	FlowLayout1
FooL	FooL
ForEtiquetado	LabeledFor
Fruta	Fruit
FuenteAgua	WaterSource
Fuga	CloudScape

Español	**Inglés**
Galleta	Cookie
GalletaChocolate	ChocolateChip
Gato	Cat
GatosYPerros	CatsAndDogs
GatosYPerros2	CatsAndDogs2
Generador	Generator
GeneradorBoolean	BooleanGenerator
GeneradorBooleanAleatorio	RandBooleanGenerator
GeneradorByte	ByteGenerator
GeneradorByteAleatorio	RandByteGenerator
GeneradorChar	CharGenerator
GeneradorCharAleatorio	RandCharGenerator
GeneradorDouble	DoubleGenerator
GeneradorDoubleAleatorio	RandDoubleGenerator
GeneradorFloat	FloatGenerator
GeneradorFloatAleatorio	RandFloatGenerator
GeneradorInt	IntGenerator
GeneradorIntAleatorio	RandIntGenerator
GeneradorLong	LongGenerator
GeneradorLongAleatorio	RandLongGenerator
GeneradorMapa	MapGenerator
GeneradorParString	StringPairGenerator
GeneradorParStringAleatorio	RandStringPairGenerator
GeneradorShort	ShortGenerator
GeneradorShortAleatorio	RandShortGenerator
GeneradorString	StringGenerator
GeneradorStringAleatorio	RandStringGenerator
Gente	People
GolpeHorror	HorrorFlick
Grafica	Glyph
GraficaCircular	RoundGlyph
GridLayout1	GridLayout1
GrupoBotones	ButonGroups
GrupoHilos1	ThreadGroup1
Gusano	Worm
Helado	IceCream
HerenciaInterna	InheritInner
Heroe	Hero
HiloClienteParlante	JabberClientThread
HiloServlet	ThreadServlet
HiloSimple	SimpleThread
Hoja	Leaf
Hola.jsp	Hello.jsp
HolaFecha	HelloDate

Español	**Inglés**
Huevo	Egg
Huevo2	Egg2
HuevoGrande	BigEgg
HuevoGrande2	BigEgg2
Humano	Human
ImplementacionesMultiples	MultiImplementation
ImprimirContenedores	PrintingContainers
InicializacionArray	ArrayInit
InicializacionStatic	StaticInitialization
InicioSesion	Logon
Inmutable1	Immutable1
Inmutable2	Immutable2
Insecto	Insect
Instrumento	Instrument
InstrumentoX	InstrumentX
Int	Int
Int1	Int1
Int2	Int2
Int3	Int3
InterfacesAnidados	NestingInterfaces
InterfacesMultiples	MultiInterfaces
InterfazI	IInterface
Interna	Inner
Interrumpir	Interrupt
Interruptor	Switch
Inverso	Reverse
Iteradores	Iterators
Iteradores2	Iterators2
ITiempoPerfecto	PerfectTimeI
ITTE	RTTI
Jabon	Soap
Jarras	Mugs
JScrollPanes	JScrollPanes
Juego	Game
JuegoMesa	BoardGame
JugueteFantasia	FancyToy
Jurasico	Jurassic
LaberintoHamster	HamsterMaze
LanzarFuera	ThrowOut
Lechuga	Lettuce
LeerOceano	OceanReading
LeerProfundidad	DephtReading
LeerTemperatura	TemperatureReading
Letal	Lethal

Español	Inglés
Libro	Book
LimitesAleatorios	RandomBounds
LimpiarMostrarMetodos	ShowMethodsClean
Limpieza	Cleanup
Linea	Line
Lista	List
Lista1	List1
ListaCajaC	CBoxList
ListaCaracteres	ListCharacters
ListadoDirectorio	DirList
ListadoDirectorio2	DirList2
ListadoDirectorio3	DirList3
ListaRaton	MouseList
Literales	Literals
LLamada1	Callee1
LLamada2	Callee2
Llueve	RainedOut
LOC_ID	LOC_ID
LOCALIZACIONES	LOCATIONS
LogicaNegocio	BusinessLogic
Logico	Bool
Lugares	Spots
MaderaViento	WoodWind
Malvado	Orc
ManipulacionBits	BitManipulation
Mapa1	Map1
MapaLento	SlowMap
MapaMultiCadena	MultiStringMap
MapeoCanonico	CanonicalMapping
MasUtil	MoreUseful
MensajePerdido	LostMessage
Menus	Menus
MenusSimples	SimpleMenus
Mes2	Month2
Meses	Months
Metal	Brass
Meteorologo	Groundhog
Meteorologo2	Groundhog2
MetodoComparacion	EqualsMethod
MetodoComparacion2	EqualsMethod2
MetodosDeExcepcion	ExceptionMethods
MiBoton	MyButton
MiClaseJava	MyJavaClass
MiDialogo	MyDialog

Español	Inglés
MIEM_APE1	MEM_FNAME
MIEM_APE2	MEM_LNAME
MIEM_ID	MEM_ID
MIEM_ID	MEM_ID
MIEM_NOM	MEM_UNAME
MIEM_ORD	MEM_ORD
MIEMBROS	MEMBERS
MiExcepcion	MyException
MiExcepcion2	MyException2
MiIncremento	MyIncrement
MiMundo	MyWorld
MiObjeto	MyObject
MiTipo	MyType
Mitologia	Weeble
MiWindowListener	MyWindowListener
ML	ML
MM	MM
MML	MML
ModeloDatos	DataModel
ModificacionPrivado	OverridingPrivate
ModificacionPrivado2	OverridingPrivate2
ModL	ModL
Molestia	Annoyance
Monstruo	Monster
MonstruoPeligroso	DangerousMonster
MostrarAddListeners	ShowAddListeners
MostrarDatosFormulario.jsp	DisplayFormData.jsp
MostrarHTML	ShowHTML
MostrarMensaje	ShowMessage
MostrarMetodos	ShowMetohds
MostrarSegundos.jsp	ShowSeconds.jsp
MostrarTiempoPerfecto	DisplayPerfectTime
Motor	Engine
Mpar	Mpair
Musica	Music
Musica2	Music2
Musica3	Music3
Musica4	Music4
Musica5	Music5
Mutable	Mutable
MuyGrande	VeryBig
NoMas	NoMore
NOMBRE	NAME
NombreL	NameL

Español	Inglés
NoSoportable	Unsupported
Nota	Note
Nota	Note
Notificador	Notifier
NotificarResumir1	NotifyResume1
NotificarResumir2	NotifyResume2
NuevoArray	ArrayNew
NuncaCapturado	NeverCaught
ObjetoClaseArray	ArrayClassObject
ObjetoSesion.jsp	SessionObject.jsp
ObjetoSesion2.jsp	SessionObject2.jsp
ObjetoSesion3.jsp	SessionObject3.jsp
Observador	Watcher
ObservadorL	WatcherL
Ocultar	Hide
OndaSeno	SineWave
OnOffL	OnOffL
OperadorComa	CommaOperator
OperadoresMatematicos	MathOps
OrdenarAlfabeticamente	AlphabeticSorting
OrdenarStrings	StringSorting
OrdenDeInicializacion	OrderOfInitialization
OrdenSobrecarga	OverloadingOrder
Ordinario	Ordinary
OyenteConteo	CountListener
OyenteEmergente	PopupListener
Pajaro	Bird
Pan	Bread
PanelTabulado1	TabbedPane1
PanelTexto	TextPane
Paquete1	Parcel1
Paquete2	Parcel2
Paquete3	Parcel3
Paquete4	Parcel4
Paquete5	Parcel5
Paquete6	Parcel6
Par	Pair
ParametrosConstante	FinalArguments
ParametrosVariables	VarArgs
PararElectoresL	StopPeekersL
PasarObjeto	PassObject
PasarReferencias	PassReferences
Pastel	Pie
Pcontenidos	PContents

Español	Inglés
PDestino	PDestination
PegarL	PasteL
Perro	Dog
Persona	Person
PersonajeDeAccion	ActionCharacter
PilaL	StackL
Pilas	Stacks
Platano	Banana
Plato	Plate
PlatoCena	DinnerPlate
Poligono	Shape
Poligonos	Shapes
PonerMesa	PlaceSetting
PRECIO	PRICE
Prediccion	Prediction
PRIMERO	FIRST
ProbarMas	TryMore
ProblemaES	IOProblem
ProductoLimpieza	Cleanser
Progreso	Progress
Prueba	Test
PruebaAcceso	TestAccess
PruebaAfirmacion	TestAssert
PruebaArrays	TestArrays
PruebaArrays2	TestArrays2
PruebaBeanExplosion	BangBeanTest
PruebaBeanExplosion2	BangBeanTest2
PruebaBiblioteca	LibTest
PruebaCama	TestBed
PruebaComparador	ComparatorTest
PruebaConjunto	TestSet
PruebaDoc	DocTest
PruebaElectorArchivo	FileChooserTest
PruebaEOF	TestEOF
PruebaEstatica	StaticTest
PruebaHerramienta	ToolTest
PruebaHilo1	TestThread1
PruebaHilo2	TestThread2
PruebaJuguete	ToyTest
PruebaListaRaton	MouseListTest
PruebaRellenar	FillTest
PruebaWhile	WhileTest
PuedeLuchar	CanFight
PuedeNadar	CanSwim

Español	Inglés
PuedeVolar	CanFly
Puerta	Door
Queso	Cheese
QuienSoyYo	WhoAmI
Rama	Branch
Rana	Frog
RastrearEvento	TrackEvent
Rastro1	Blip1
Rastro2	Blip2
Rastro3	Blip3
Rastros	Blips
Raton	Mouse
RealizarPrueba	Tester
RealizarRastreo	TrackingSlip
RealmenteNoMas	ReallyNoMore
Rearrancar	Restart
Receptor	Receiver
RecuentoAnimalDomestico	PetCount
RecuentoAnimalDomestico2	PetCount2
RecuentoAnimalDomestico3	PetCount3
RecuentoPalabra	WordCount
RecursividadInfinita	InfiniteRecursion
Redireccionar	Redirecting
Referencias	References
ReglasServlets	ServletsRule
Relanzando	Rethrowing
RellenarArrays	FillingArrays
RellenarListas	FillingLists
RendimientoConjuntos	SetPerformance
RendimientoListas	ListPerformance
RendimientoMapas	MapPerformance
Resumidor	Resumer
Retornar	BackOn
Retrollamadas	Callbacks
Rueda	Wheel
SalvarL	SaveL
Secuencia	Sequence
SeguirSesion	SessionPeek
Semilla	Seed
Separacion	Separation
Serpiente	Snake
ServidorMultiParlante	MultiJabberServer
ServidorParlante	JabberServer
ServidorTiempoExacto	ExactTimeServer

Español	Inglés
ServidorTiempoRemoto	RemoteTimeServer
ServirUnParlante	ServeOneJabber
SiempreFinally	AlwaysFinally
Silla	Chair
Sincronizacion	Synchronization
SistemaDAC	CADSystem
Sobrecarga	Overloading
SobrecargaPrimitivo	PrimitiveOverloading
SoloDatos	DataOnly
SoloLectura	ReadOnly
Sopa	Soup
StaticExplicito	ExplicitStatic
SubTareaSeparada	SeparateSubTask
Suspender	Suspend
SuspenderResumir	SuspendResume
SuspenderResumir1	SuspendResume1
SuspenderResumir2	SuspendResume2
Suspendible	Suspendable
T1	T1
T1A	T1A
Tabla	Table
Tarta	Cake
TELEFONO	PHONE
Teletipo	Ticker
Teletipo2	Ticker2
Tenedor	Fork
TermostatoDia	ThermostatDay
TermostatoNoche	ThermostatNight
TicTacToe	TicTacToe
TiempoExacto	ExactTime
TiempoPerfecto	PerfectTime
TiempoPerfectoHome	PerfectTimeHome
TIPO	TYPE
TipoComp	CompType
TITULO	TITLE
TML	TML
TodosOperadores	AllOps
Tomate	Tomato
TrabajarCualquierModo	WorksAnyway
TrampaRaton	MouseTrap
Transformar	Transmogrify
Triangulo	Triangle
ULTIMO	LAST
UsarObjetos	UseObjects

Español	Inglés
UsarObjImpl	UseObjImpl
Utensilio	Utensil
Util	Useful
Vainilla	Sundae
Valor	Value
ValoresAleatorios	RandVals
ValoresIniciales	InitialValues
ValorInt	IntValue
Vampiro	Vampire
Ventana	Window
Viento	Wind
VientoX	WindX
Villano	Villain
Visita	Caller
VocalesYConsonantes	VowelsAndConsonants
Volcador	Dumper
VolcadorBean	BeanDumper
WhileEtiquetado	LabeledWhile
Yema	Yolk
ZebraVerde	GreenZebra

Índice

B

C

E

F

G

J

M

N

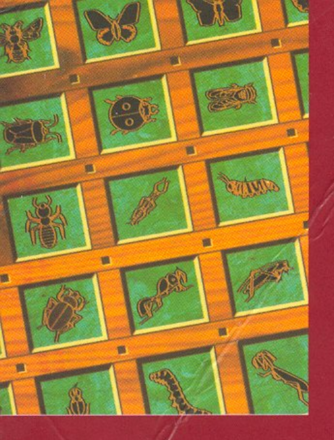
O

P

R

S

T



2ª Edición

Piensa en Java

Bruce Eckel

¡El libro de JAVA más premiado!

PIENSA EN JAVA segunda edición, introduce todos los fundamentos teóricos y prácticos del lenguaje JAVA, explicando con claridad y rigor no sólo lo que hace el lenguaje sino también el porqué.

Escrito para "enseñar" el lenguaje, es un libro que sirve para iniciarse en JAVA y para llegar hasta un nivel avanzado, con gran número de ejemplos y ejercicios de programación.

Características

- Presenta los conceptos gradualmente, para que el lector pueda asimilarlos conceptualmente.
- Proporciona una presentación cuidadosa y secuencial de las características y una introducción breve a modo de descripción.
- Da pistas necesarias para comprender el lenguaje, remarcando la información relevante heredada, pero sin confundir al lector con hechos precedentes que el programador nunca necesitará saber.
- Provee al lector de fundamentos sólidos para que pueda entender cuestiones importantes, y para que reaccione ante las dificultades que se plantean durante el aprendizaje del lenguaje
- Esta centrado en la versión 2 de Java.
- Enseña las propiedades del lenguaje de un modo independiente de la plataforma.
- Explica los principios básicos de la orientación a objetos y cómo se aplican a Java.
- Incluye más de 300 programas Java y más de 15.000 líneas de código.



En el CD Rom que acompaña al libro se puede encontrar el código fuente, una versión electrónica del texto, en inglés, y un completo seminario multimedia, también en inglés, *Thinking in C: Foundations for C++ & JAVA.*

Pearson Educación

www.pearsoneducacion.com

ISBN 84-205-3192-8

9 788420 531922